A PRODUCTIVE RESPONSE TO LEGACY SYSTEM PETRIFICATION

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

by
Anthony Philip James Lauder
August 2001

Abstract

Requirements change. The requirements of a legacy information system change, often in unanticipated ways, and at a more rapid pace than the rate at which the information system itself can be evolved to support them. The capabilities of a legacy system progressively fall further and further behind their evolving requirements, in a degrading process termed *petrification*. As systems petrify, they deliver diminishing business value, hamper business effectiveness, and drain organisational resources.

To address legacy systems, the first challenge is to understand how to shed their resistance to tracking requirements change. The second challenge is to ensure that a newly adaptable system never again petrifies into a change resistant legacy system. This thesis addresses both challenges.

The approach outlined herein is underpinned by an agile migration process - termed *Productive Migration* - that homes in upon the specific causes of petrification within each particular legacy system and provides guidance upon how to address them. That guidance comes in part from a personalised catalogue of *petrifying patterns*, which capture recurring themes underlying petrification. These steer us to the problems actually present in a given legacy system, and lead us to suitable antidote *productive patterns* via which we can deal with those problems one by one.

To prevent newly adaptable systems from again degrading into legacy systems, we appeal to a follow-on process, termed *Productive Evolution*, which embraces and keeps pace with change rather than resisting and falling behind it. Productive Evolution teaches us to be vigilant against signs of system petrification and helps us to nip them in the bud. The aim is to nurture systems that remain supportive of the business, that are adaptable in step with ongoing requirements change, and that continue to retain their value as significant business assets.

Acknowledgements

I count myself in nothing else so happy
As in a soul remembering my good friends.
- William Shakespeare, *Richard II* II, 3

Firstly, deepest thanks to Dr. Stuart Kent, my supervisor, who saw me through more than three years of very challenging work. Stuart is truly inspirational. Indeed, it was Stuart's own work that initially inspired me to abandon a well-paid consultancy career in Luxembourg, and return to the UK to pursue a Ph.D. program. Thanks must also go to both EPSRC and EDP, for funding the research underlying this thesis. Without their generous financial support, this work would not have been possible.

I owe a great deal of thanks to the guys "in the trenches" at EDP, who provided a concrete setting for this research. Especial thanks must go to John Corner, Paul Gledhill, and Neil Tomlinson, who constantly made me justify the practicality of my "book learning", as they (jokingly) called it. In particular, John and Paul read substantial fragments of an early draft of this thesis and were often deliciously cruel in their scrutiny, prompting some substantial reworking on my part. Bob Doncaster proofread the entire thesis extremely thoroughly and provided invaluable feedback, particularly on errors grammatical and speeling mistoks. Thanks also to Roy Leigh and Richard Jowitt who kept my head out of the clouds by continually reminding me of the "commercial imperative" to deliver better products into customer hands "yesterday".

David Seed, a professional Pharmacist, was particularly helpful with analogies between medical diagnosis and drug prescription and petrifying pattern recognition and productive pattern deployment. Thanks for the helpful conversations, the British National Formulary, and above all, the great wine. All of which were highly stimulating.

On a personal level, I want to thank Maxine for being a wonderful, loving companion. Thanks too to Amber, my beautiful daughter, sadly so far away, and welcome to Beatrice, now just a few weeks old, but already bringing so much joy into my life.

Finally, to Tracey, my sister: I miss you.

Contents

List of Figures

Chapter 1

Introduction

If circumstances lead me, I will find
Where truth is hid, though it were hid indeed
Within the centre.
- William Shakespeare, *Hamlet* II, 2

1.1. Legacy Systems

This thesis deals with legacy systems. Bennett [Bennett, 1995] defines legacy systems rather loosely as "large software systems that we don't know how to cope with but that are vital to our organization". This definition brings out some important points that need elaborating:

Firstly, legacy systems tend to be large. With size tends to come complexity. As Bennett himself points out, in general "small programs are not difficult to maintain" [ibid.], but large programs are.

Secondly, legacy systems are usually vital to our organisation; they tend to support (or, rather, fail to adequately support) core business processes via which an organisation operates. Business processes necessarily evolve over time, thus mandating ongoing change in the requirements for the information systems supporting them. Unfortunately, and herein lies the problem, legacy systems significantly and continually resist such change. That is, a legacy system progressively fails to support an organisation's (evolving) business processes, so that its value to the business diminishes over time.

A legacy system, then, becomes increasingly less of a business asset and more of a business liability. This bring us to the third point in Bennett's definition: "we don't know how to cope with" legacy systems, and hence we don't know how to prevent them from stifling organisational effectiveness. This is the central issue: learning how to deal with legacy systems so that they no longer hold the business back. We need to work out how to tackle legacy systems' resistance to change so that they are no longer business liabilities but rather remain valuable business assets, evolving with, and continually supporting (rather than hindering) the changing business processes underpinning organisational effectiveness. This thesis addresses that issue.

1.2. We Once Had an Excuse

Historically, diminishing business value may have been unavoidable: there was a time when we were so severely constrained by technology (tiny storage capacities, programming in octal using toggle switches, etc.) that it was hard enough to deal with today's requirements, never mind tomorrow's. As technology began to improve, though, anecdotal evidence implies that the problem shifted to a widely held belief that any given piece of software would be short lived. Thus, it was often argued that there was no point thinking too far ahead into the future, since, by the time a system had outgrown its requirements, it would already have been rewritten many times over. This, of course, proved to be a gross underestimation of the potential durability of software. For example, recent panics over Y2K (a highly predictable requirements change) rather publicly disproved the presumption of a short life span for a great many “aging” software systems upon which we are still very much dependent.

1.3. But Now We Don't

Nowadays, we have to accept that a given software system will probably live for a long time, and that its requirements will change frequently throughout its life. Furthermore, those requirements are likely to change in highly unpredictable ways, invalidating many of the assumptions that were held by the system's original designers. For example, support for the Euro is a widely quoted recent requirements change that caught out many unwary software developers, whose systems significantly resisted support for multi-currency.

1.4. Accepting Change

If we continue to deny that requirements will change, particularly in unforeseen directions, and strive to pin down information system requirements up-front, with essentially stable system architectures supporting them, we are in danger of building the legacy systems of the future. Unfortunately, as we shall see in later chapters, many prevalent development methodologies - even highly popular ones - encourage us to do just that.

To retain the business asset value of information systems we need to change our stance towards requirements change. Thus, rather than fearing change, we need to accept that change is natural and inevitable, and ultimately manageable and even beneficial. Armed with this positional shift, we can then begin to tackle the legacy systems that hold us back, and prevent further legacy systems from emerging in the future.

1.5. Productive Migration

The overall aim of this research has been to uncover practices with which we can successfully transform (migrate) legacy systems into systems that maintain pace with ongoing business process (i.e. requirements) change. Rather than propose a naïve "one size fits all" solution, the intent has been to derive an approach that is highly adaptable to the needs of diverse legacy system migration projects. The result is a migration process, termed *Productive Migration*, which remains agile amid changing contextual needs. In particular, Productive Migration is intended to be rigorous enough to help steer a successful migration effort, yet also adaptive enough to allow its continual reshaping to the unfolding needs of diverse legacy systems and the migration teams that address them.

We will see in a later chapter that, ultimately, much of a legacy system's resistance to supporting ongoing business process change stems from its architectural misalignment with the needs of the business processes it is meant to support. Productive Migration, then, aims to realign a system's architecture with the needs of those business processes.

The most significant externally visible indicator of a legacy system is a backlog of pending business process changes whose support the legacy systems resists. We can use this backlog to drive the migration process. More specifically, we will see that by prioritising the items on the backlog according to relative business value, we can perform Productive Migration incrementally and profitably. During each Productive Migration increment, we address the currently highest priority items from the backlog, progressively loosen the legacy system's architectural resistance to that business value, inject support for the business value into the system, and finally deliver another release back into customer hands.

1.6. Productive Evolution

Over a series of incremental releases, then, Productive Migration gradually chips away at the legacy system's resistance to its backlogged requirements, until the legacy system is considered sufficiently adaptable with respect to those requirements. Once a legacy system has been migrated to an adaptable system, we want to prevent it from again resisting requirements change in the future. That is, we want to ensure that further evolution of the system - in the light of ongoing change in the business processes it supports - occurs gracefully. This is important, since an information system that maintains pace with business process change ultimately permits a deliberate co-

evolution of both the information system and the business processes it supports. That is, the information system becomes a business asset, rather than a business liability.

Unfortunately, petrification has a nasty habit of sneaking back into a newly adaptable system, as the slow drip effect of a succession of individually innocuous hacks gradually erodes system adaptability from within. It is only much later, when progressive erosion finally manifests itself externally through an unmistakable backlog of resisted requirements, that the real, and literally petrifying, impact of such erosion becomes all too clear. By then, of course, it is too late; we will again be back where we started, with a legacy system resistant to, and thus stifling, support for ongoing business process change.

We need, therefore, to be vigilant against the types of *dis*graceful evolution that lead to change resistance, and instead continually steer evolution towards systems that remain adaptable. To achieve this, we can appeal to a process of *Productive Evolution*, which, like Productive Migration, is a learning process, low on ceremony and high on productivity. Productive Evolution focuses upon preventing change resistance from occurring in the first place.

We will see later in the thesis that Productive Evolution in practice is not much different from Productive Migration. Importantly, the similarity in practices of Productive Migration and Productive Evolution significantly simplifies the adoption of a mixed approach, wherein we coordinate both Migration and Evolution work. At a high level of abstraction, the only real difference between Productive Migration and Productive Evolution is that with Migration we know that the system significantly resists the requirements on the backlog, whereas with newly emerging requirements we are unsure what the level of resistance will be. Thus, we shall see that for a new requirement, Productive Evolution begins with a brief “toe dipping” exercise to measure the system’s current level of resistance to that requirement. If the resistance is minimal, we may choose to inject support for the requirement immediately by continuing the current Productive Evolution increment. If, on the other hand, resistance is significant, we can choose to place the requirement on the backlog for subsequent Productive Migration work.

1.7. Petrifying vs. Productive Patterns

A significant proportion of the research underpinning this thesis has concentrated upon the ongoing cultivation of a catalogue of patterns of migration expertise. These both support and learn from Productive Migration and Productive Evolution in practice. Such

patterns help us to recognise change resistance, which we term *petrification*, to migrate legacy systems to adaptable systems, and then to keep those systems permanently adaptable in response to future change.

Our first concern is to uncover patterns - termed *Petrifying Patterns* - that contribute to the erosion of system adaptability and hence are potential sources of petrification. We find such patterns primarily by empirical study of known legacy systems.

Petrifying Patterns are contra-indicated patterns, in the medical sense that they are generally "bad medicine" with undesirable side effects. We view them as basically "damaging experiences" best avoided, and whose presence indicates that a system's long-term health (i.e. adaptability to requirements change) is in question.

By looking at systems (or, more usually, parts of systems) that have successfully embraced change, we are able to extract the essence of such accommodation as curative *antidotes* to petrification. We can then administer these antidotes where relevant symptoms of change resistance (i.e. Petrifying Patterns) are seen. Such antidotes are termed *Productive Patterns*, since they tend to enable us to remain productive, prevent petrification, and keep pace with ongoing requirements change. The careful application of suitable Productive Patterns can give new life to a legacy system, taking a once change resistant system and making it adaptable again.

1.8. Making It Personal

It is vitally important to recognise that Productive Migration and Productive Evolution are independent of the specific Petrifying Patterns and Productive Patterns that are presented in the appendices. Although the patterns in this thesis are highly relevant to the legacy systems under study, their relevance to other legacy systems can only be determined in the specific context of those other legacy systems. Consequently, this thesis advocates a "grow-your-own" approach wherein each organisation, or even each project team, cultivates their own personalised catalogue of patterns that are deeply relevant to their own needs.

A cultivated and personalised pattern catalogue continues to capture and evolve with its owners' own migration experience, growing with them, and adapting with their changing circumstances. Such a pattern catalogue can help guide its caretakers in a tailored migration of their own legacy systems into suitable adaptable replacements, by addressing the very real patterns underpinning the change resistance prevalent in the specific legacy systems being migrated. Importantly, a well-cultivated pattern catalogue keeps a migration team on track, helping them recognise when they are straying off in

dubious directions. In particular, patterns help protect migration teams from tempting courses of treatment which ultimately prove inappropriate for the specific needs of the legacy systems upon which they are currently focused.

Once migration is complete, the cultivated pattern catalogue continues to support vigilance against petrification creep so that newly adaptable systems are protected from stealthy degradation back to legacy. That is, the information systems remain a valuable business asset, and no longer degrade into a business liability.

1.9. Contributions

The particular contributions of this thesis, then, are seen to be:

- A new focus on *dis*graceful architectural petrification through waves of hacking, in response to unexpected change, as the fundamental cause (rather than age or technology) of business process change resistance in legacy systems.
- A novel agile methodology (Productive Migration) with which to de-petrify a legacy system by progressively rearchitecting that system in step with an evolving understanding of the commonality and variability needs to the changing business processes it is intended to support. This is followed by an ongoing process of Productive Evolution, which retains system adaptability in the light of continual business process change, and thus facilitates coordinated co-evolution of business processes and the information systems supporting them.
- A redirection away from speculative universal cure-alls, to the cultivation of personalised catalogues of Petrifying Patterns, with which to guide Productive Migration in the identification of the very specific problems present in individual legacy systems. This is complemented by the cultivation, cataloguing, and deployment of appropriate targeted responses (Productive Patterns), which are seen as patterns manifest in the type of adaptable system towards which we strive. In simple terms, Productive Migration strives to replace Petrifying Patterns with corresponding Productive Pattern antidotes, and Productive Evolution strives to prevent Petrifying Patterns from creeping back into the system.
- A catalogue of both Petrifying and Productive Patterns mined and refined through empirical study, as an inspirational starting point for the elaboration of similar catalogues within the context-specific requirements of other legacy system migration and evolution efforts.

1.10. Research Context

The research underlying this thesis was one of about thirty projects within the EPSRC managed research program entitled Systems Engineering for Business Process Change (SEBPC) coordinated by Peter Henderson. Each of these projects comprised academic research grounded in a commercial setting through industrial partnership.

Within the particular project from which this thesis derived, I was the sole investigator, with Stuart Kent acting as supervisor. My work on this research project provided both the stimulus and theory for Productive Migration, Productive Evolution, and Petrifying and Productive Patterns as described in later chapters. To provide an empirical “test bed” for the efficacy of that theory, I have been working hand in hand with a industrial partner, Electronic Data Processing PLC (EDP), since March 1998.

EDP was formed in the 1960s to provide bureau computer services to commercial organisations. Throughout the 1970s there was a strong shift of the business away from bureau services to the provision of bespoke information systems. This was followed by a further shift in the 1980s to the provision of standardised, albeit highly configurable, packaged solutions. By the early 1990s, EDP was well established as a provider of both shrink-wrapped business software and tailorable application frameworks.

Throughout much of its thirty-five year history, EDP has been developing commercial software primarily for the sales and distribution industry. This is a large market, and the provision of software to that market is intensely competitive. EDP's products have proven particularly popular, and this success has enabled the company to purchase a number of direct competitors over the years. In several cases the motivation was the acquisition of an established customer base rather than the particular software solutions offered by the competitor. Where possible, EDP has made the commercial decision to switch that customer base over to EDP's own software solutions, providing various incentives to do so. In some cases, however, direct product substitution has proven impossible, principally where there is considerable functionality in an ex-competitor's products that is not present in those offered by EDP and vice versa.

This history of business acquisitions has left EDP with a problem: Both EDP and the acquired ex-competitors have been delivering software solutions to their chosen market for many years. Throughout that time their products have slowly degraded so that they are now, for the most part, perceived as legacy systems. That is, these products have undergone a gradual process of petrification, leaving them so inflexible that supporting the changing needs of the sales and distribution market has become extremely

challenging. To remain competitive in today's increasingly demanding marketplace, EDP has recognised the need to tackle its accumulated legacy liability.

Thus far EDP has adopted an essentially just-in-time tactical rewrite approach to the legacy system problem: every few years a team of developers has been dispatched to develop a replacement system from scratch. Unfortunately, as software applications have become increasingly sophisticated (now involving up to five million lines of code each at EDP), this redevelopment effort has become progressively prohibitively expensive[1]. Furthermore, although this approach has certainly helped temporarily to eliminate the symptoms of legacy systems, it has never emphasised the identification of and vigilance against the underlying causes of petrification.

EDP has learned that by continuing to follow its current tactical approach, it can be almost certain that today's new replacement systems will become tomorrow's legacy systems. Consequently, EDP has committed to, and has been pursuing through this research project, a shift away from tactical rewrites towards the strategic software migration and evolution processes outlined here[2].

Throughout collaboration with EDP, numerous Petrifying Patterns have been uncovered in a variety of legacy systems owned by the company, and appropriate antidote Productive Patterns have been cultivated and administered in response. This thesis presents (in the appendices) many of the patterns mined at EDP, and provides a pattern-catalogue-oriented roadmap with which other organisations can begin to tackle the

---

[1] Of course, Gallagher would have us believe that 100% of development costs are attributable to maintenance, since "there is no such thing as software development; development is a special case of maintenance" [Gallagher, 2001]. Most authorities, though, take a less radical view and quote maintenance costs of approximately 60%, referring primarily to studies from the 1970s (see, e.g., [Pigoski, 1997]). Interestingly, [Swanson, 1976] shows that the percentage has actually gone up since then, with contemporary studies putting the figure for the 1990s as high as 95%! One possible implication is that recent technical advances have made new development cheaper, but have been less effective in reducing the cost of maintenance. Another possibility is that the level of maintenance work compared to new development work has increased. Indeed, it appears that as the business world has become increasingly competitive, the rate at which businesses are forced to change in order to remain competitive has spiralled. This may well have resulted in a corresponding surge in the demand for software requirements change, so that developers are increasingly "swamped" in a flood of change requests, with a corresponding increase in associated costs.

[2] Unfortunately, since the level of ongoing requirements change is not expected to decrease in the future, it is not clear that switching from a wholesale rewrite approach will actually save much money (although efficiency gains from the practices described in this thesis may make some difference). Having said that, our expectation (and our recent experience at EDP) is that we can at least amortise that cost over time if we make system "maintenance" an intrinsic part of development throughout the life of a product. We can, however, expect to make a greater impact in terms of time-to-market; by keeping the system responsive to change throughout its life, we can negate the need for lengthy late-in-the-day rewrites, thus reducing the time lag between change request and delivery. The expectation (again supported by our recent experience) is that the resultant satisfaction of customer requirements can then fund further ongoing "maintenance" work.

legacy systems that hold them back. A cultivated pattern catalogue captures growing migration and evolution expertise, and thus complements Productive Migration and Productive Evolution in action.

1.11. Authorship

Helping EDP tackle their own legacy systems has provided an unforgiving "in the trenches" reality check for some of my early and naïve assumptions about legacy systems, and hence "the practice" has helped to tighten up the theory considerably. I am extremely grateful, then, to the people at EDP for their willingness to try out my evolving theories, and for their practical contributions to my understanding of "what really works in the field".

Throughout collaboration with EDP I have been careful to ensure that responsibility for reflecting upon the nature of legacy systems, and evolving Productive Migration and Productive Evolution in response, is entirely mine, so that, in accordance with PhD requirements, I can claim the work – and its contribution to knowledge - to be my own.

Turning to the Petrifying and Productive Patterns that support Productive Migration, I can only claim as my own the initial inspiration for them and their underlying theory (see chapter 13). The patterns themselves (see appendices), on the other hand, have emerged primarily from reflective observation of the work of the EDP teams with which I have worked closely. Thus, I have to consider the formation of many of the patterns to be a joint work between myself and colleagues at EDP[3]. Having said that, recognition of them as patterns, their embellishment with lessons from other research, and their "tightening up" into pattern form is entirely my own.

1.12. The Remaining Chapters

The remaining chapters within this thesis are as follows:

2. **Business Process Change (Knowing the History):** Our first concern is to understand why requirements change. Since the requirements for an information system stem from the business processes it supports, this question is basically asking why *business process change* occurs. We address this question in **chapter 2**, where

[3] There are exceptions to this, for example the Dynamic Pluggable Factory and the Explicit Protocol Reflection patterns are entirely my own work, and were introduced by me to EDP as new practices. However, other patterns (e.g., Legacy Customers, and its antidote A Spoonful of Sugar) were in use at EDP prior to my arrival (although admittedly they were not recognised as patterns). The stimulus and spirit (if not the detail) of the majority of patterns in the appendices, though, clearly emerged from close collaboration with EDP colleagues throughout our migration work, so that their "invention" can only rightly be considered shared.

we will see, through a review of relevant literature, that business process change is intrinsic to business survival in an increasingly competitive world.

3. **Legacy Systems (Acknowledging the Problem):** Next, we need to recognise that legacy systems do not support changing business processes adequately. Legacy systems, then, are a potential business impediment, rather than a business asset. In **chapter 3**, we briefly survey existing legacy system literature, homing in on a definition that casts legacy systems not as a problem of old age or dead technology but of *change resistance*.

4. **Petrification (Issuing a Health Warning):** In **chapter 4** we uncover how legacy systems actually become change resistant. This requires an understanding of what typically occurs in response to ongoing business process change, and what that typical response does to an information system's architecture. This is essentially recognition that adaptable systems degenerate (through successive waves of *hacking*) into *petrified* legacy systems, whose *accidental architectures* resist the reflection of ongoing requirements change and hence fail to support businesses adequately.

5. **Developer Maturity (It's Not For the Faint Hearted):** Tackling legacy systems is hard work; it requires developers to be highly skilled and fearless in their migration work. In **chapter 5** we explore three levels of developer maturity, and observe that developers need to work at the *highest level of maturity* (Adaptation) to tackle and subsequently prevent system petrification effectively.

6. **Predictive Methodologies (An Unhealthy Lifestyle):** Many prevalent development methodologies strive to either pin down requirements up front, or home in on stable architectures with essentially fixed points of requirements variability. These *predictive methodologies*, reviewed and discussed in **chapter 6**, effectively encourage legacy systems by preventing us from embracing unanticipated requirements change.

7. **Agile Methodologies (A Guide to Healthy Living):** To deal effectively with legacy systems, and to prevent them in the future, we need to appeal to practices that embrace change. *Agile methodologies* do just that. As we shall see in **chapter 7**, they inspire us to be light on our feet, to appeal to humans rather than mechanistic processes, and embolden us to continually change our beliefs and practices when they are hindering rather than helping us.

8. **Productive Migration (Responding to the Epidemic):** *Productive Migration*, introduced in **chapter 8**, is a novel agile response to petrification that focuses on high-productivity. It mitigates risk by adopting a piecemeal business-value-driven migration strategy, and also addresses human trepidation during migration work by appealing to practices that respect the first law of medicine: "do no harm".
9. **Migration Marathons (Prioritising Urgent Cases):** In **chapter 9** we will see that Productive Migration manages the sheer size of legacy system migration by addressing it incrementally. Each increment, termed a *Migration Marathon*, is scoped and prioritised according to both the urgency of resisted business value and the readiness of developers to accommodate that business value.
10. **Migration Runs (Leave it to the Specialists):** Migration Marathons are usually decomposed into independent constituent *Migration Runs*, each focused upon one area of missing business value. **Chapter 10** reveals how separate Run teams work as far as possible in parallel, thus maximising progress within a given Marathon.
11. **Migration Sprints and Bursts (Monitoring the Patient's Progress):** We often break Runs down into smaller sub-projects, termed *Sprints and Bursts*, revealed in **chapter 11**. These give concentrated short term focus, allowing the team to demonstrate mid-Run progress, catch mistakes early, and take corrective action if things aren't going as expected.
12. **Productive Evolution (Prevention is Better Than Cure):** Having successfully tackled a petrified system, we want to ensure that today's adaptable systems do not degrade into tomorrow's legacy systems. This is essentially a matter of vigilance: keeping abreast of requirements change, and looking out for early signs of petrification and nipping them in the bud. This process of *Productive Evolution* is the focus of **chapter 12**.
13. **Patterns to the Rescue (Expert Diagnosis and Prescription):** As we gain experience with legacy system migration, we see two types of pattern emerge. *Petrifying Patterns* encapsulate symptoms and causes that contribute to the erosion of adaptability, and these help us to identify the sources of petrification in individual legacy systems. From systems (or, more usually, parts of systems) that have successfully embraced change we can mine corresponding curative *antidotes*. These *Productive Patterns* bring features and benefits that can help us loosen the symptoms and causes of petrification, and thus reintroduce system adaptability. As described in **chapter 13**, cultivating our own pattern catalogues enables us to

personalise Productive Migration and Productive Evolution according to both our own experience and the specific needs of the actual legacy systems with which we are faced.

14. **Looking Ahead (Where Do We Go From Here?):** The penultimate chapter, **chapter 14**, reflects upon appropriate future work in continuation of this thesis, and warns how the initially "obvious" value of additional tool support is less obvious than it first appears.
15. **Contributions (So, What's New About That?):** The final chapter, **chapter 15**, reflects upon how an empirical setting proved rather sobering, keeping initial enthusiasm for essentially intellectually interesting but ultimately somewhat misguided ambitions in check. Armed with a recast focus upon what the real legacy system challenges are, we enumerate the particular contributions of this thesis towards addressing those challenges.

A. **Petrifying Pattern Catalogue (Learning From Our Mistakes): Appendix A** captures a substantial number of Petrifying Patterns mined during migration work at EDP. It took a long time to cultivate this catalogue, and its shape has changed continually throughout this time. We have learned that it neither possible nor desirable to "pin a catalogue down" once and for all; to remain useful a pattern catalogue needs to continually evolve with our growing expertise, or else risk stagnation. Appendix A, then, contains a snapshot of Petrifying Patterns taken towards the end of this research project. EDP's pattern catalogue will certainly continue to evolve beyond that presented in appendix A, as they gain more and more migration expertise.

B. **Productive Pattern Catalogue (Learning From Our Successes): Appendix B** captures a snapshot of corresponding antidote Productive Patterns to the Petrifying Patterns contained in Appendix A. It is to precisely these patterns that EDP has appealed throughout this research project, and continues to appeal (and evolve) in the migration of their petrified legacy systems to adaptable replacements.

Chapter 2

Business Process Change

This world is not for aye, nor 'tis not strange
That even our loves should with our fortunes change.
- William Shakespeare, *Hamlet* III, 2

2.1. Preview

The economist Warren Bennis was probably the first person to write about the relevance of business process change to commercial competitiveness [Bennis, 1966]. More recently, the Business Process Reengineering (BPR) community (starting with [Hammer, 1990]) has stimulated a widespread acceptance of the potential benefits that business processes change can bring. Whether or not we want business process change to occur, there is now emerging evidence that such change is probably unavoidable anyway (see, e.g. [Morgan, 1996; Mitleton-Kelly and Papaefthimiou, 2000b; Mitleton-Kelly and Papaefthimiou, 2001]).

With growing acceptance of business process change, it is becoming increasingly clear that information systems must evolve in their support of changing business processes if they are to remain of value to (rather than being detrimental to) an organisation [Henderson, 2000b; Henderson, 2001]. Now, early writings on BPR (e.g., [Hammer and Champy, 1993; Davenport, 1993]) cast IT as a potential enabler of business process change. In practice, though, existing information systems often have the opposite effect; "A major problem is the extent to which IT is a disabler of business process change, when in fact it should be an enabler" [Henderson, 2000a].

As noted in chapter 1, information systems that resist business process change are termed *legacy systems*. To enable organisations to remain competitive amid commercial pressures, and thus continually adapt their business processes accordingly, we need to tackle the legacy systems that stifle business process change and that consequently hold those organisations back.

To uncover how and why legacy systems resist business process change, we need first to explore more deeply the nature of business processes - examining what they are, why they exist, and why they change over time. In this chapter, then, we turn to business literature, to sociology, and to Scandinavian perspectives on organisational behaviour to

understand why business process change is endemic, and we take this as a starting point with which to uncover why the inability of a legacy system to adapt to that change can be a serious threat to organisational competitiveness. In the next chapter, we will discuss various strategies for dealing with legacy systems to cope with business process change, and identify an important, unresolved challenge, which this thesis addresses.

2.2. Organisational Goals

Organisations exist, ultimately, to fulfil their ambitions. The particular ambitions vary from organisation to organisation. For some organisations an overriding ambition may be to maximise profit in the shortest possible timeframe, whereas for others there may be little profit motive whatsoever (charities, perhaps). For most organisations, there will be multiple ambitions, with varying lifetimes, and with varying levels of clarity. Frequently, of course, there will be tensions between competing ambitions, and these tensions need to be balanced. Maximising profit, for example, generally conflicts (at least in the short term) with investing in research and development for the next generation of leading-edge products or services. It is the balancing of such conflicts that tempers organisational wants into organisational needs, which are themselves expressed as *organisational goals*.

Organisational goals are seen as fundamental to organisational existence. Indeed, the term *organisation* is used here to denote a collection of mutually recognised individuals bound by, and collaborating in fulfilment of, common long-term goals [Lauder and Lind, 1999]

2.3. Business Processes

In order to achieve its goals, an organisation constructs (either implicitly or explicitly*)* *business processes* whose outcomes contribute to those goals. We can think of a business process as a semi-structured collection of activities [Ould, 1995] contributing to the satisfaction of one or more business goals[4]. The more an activity contributes to the goals of an organisation, the higher its value to that organisation. The more a given activity detracts from the goals of an organisation, the more its value is reduced. Constraints placed upon these activities are often termed *business rules*. Information

[4] It has proven difficult to produce a "tighter" definition of the term business process than this one. Harrison, for example, has noted that much is written about business processes but little consensus has been attempted in defining what they are [Harrison, 1995]. His survey [ibid.] of seven published definitions concludes "Processes are variously defined in terms of activities, tasks and functions or steps which may lead to business success, output of value to a customer or simply a result." Harrison bemoans "this lack of clarity", asking "If you cannot get the basics right, then how can you do the more difficult things?"

systems tend to support business processes, at least in part, by enforcing such business rules.

2.4. Collaboration

Few, if any, organisations are able to satisfy their goals in complete isolation, and hence organisations tend to collaborate with one another. When an organisation wishes to collaborate in pursuit of its own goals, it usually has a variety of opportunities for collaboration, with other organisations offering a variety of possibilities for the exchange of goods, services, or other assets deemed to be of value. An organisation will see a particular collaboration as beneficial, and thus worth pursuing, only if that collaboration contributes to the satisfaction of its own goals more than it detracts from them. Consequently, each organisation must be willing to entice potential collaborators into a collaboration by offering goods or services (or other assets) at a "price" that is tempting. An organisation strives to balance the tension between what it must contribute to others and what it receives in return in order to determine whether or not a given collaboration is valuable, and thus worth participating in.

2.5. Value Comparison

To decide which collaboration opportunities offer the best value, a collaborator must be able to compare their potential outcomes. The relative value [Emerson, 1987] each organisation places on the goods or services offered by another in a collaboration determines their relative bargaining power and the underlying exchange rate [Marsden, 1987]. The determination of an exchange rate (how much to give up for what is returned) is guided both by rational calculation (which can, potentially, be automated) and, importantly, by emotional judgment (which is inherently human) based upon long-term processes of conditioning [Emerson, 1987] through histories of successful (and unsuccessful) collaboration.

2.6. Remaining Competitive

To entice collaboration, an organisation's business processes typically focus upon balancing the delivery of best value to all of its collaborators, so long as this is not detrimental (in the long term) to the pursuit of the organisation's own goals. To achieve this, an organisation must be able to judge not only what contributes or diminishes value for themselves, but also what contributes or diminishes value for its potential collaborators. Collaborators are often able to make reasonably good guesses about each other's preferences and values since they frequently share similar cultures and backgrounds [Friedman, 1987]. Additionally, much of the knowledge required to make

such judgments is both sought out and revealed in the course of collaboration. Identifying the preferences and values most cherished by a collaborator should enable a reasonable prediction of that collaborator's actions, and thus indicate to us the value potential of those actions, and the actions we should take ourselves to satisfy that collaborator's own goals. It is via the continual crafting of value-laden business processes (and, indeed, products and services [Mitleton-Kelly and Papaefthimiou, 2000b]), in the light of such reflection, that an organisation remains competitive.

2.7. Business Process Change

Competition and a changing business climate can reduce perception of the relative value of the outcome of a given business process, thus (possibly) tempting a collaborator to consider alternative collaboration opportunities. To remain competitive, therefore, an organisation must remain responsive to such fluctuations of perceived value and *adapt their business processes* to deliver permanently highly valued outcomes. This mandates continually listening to current and potential collaborators' needs (goals) and either fine-tuning existing processes (the TQM approach [Deming, 1986]) or completely reengineering business processes (the BPR approach [Hammer, 1990]) in response to those evolving needs[5]. *Business process change*, then, is a response to the need to remain competitive by perpetually striving to deliver perceived high value to all concerned.

2.8. Unexpected Business Process Change

This last point is an important one; organisations need to be continually sensitive to and responsive to the evolving preferences and values of would be collaborators. Since we cannot fully predict shifts in collaborator interest, competitor offerings, and the availability of supporting technology, we cannot fully predict the types of business process change that will be necessary in response to those shifts. Thus, although experience and cultural commonalities will help us to anticipate much expected change (for which we can plan ahead), unexpected shifts in the business environment force us to accept *unanticipated change* in terms of how to continually and effectively entice

[5] Actually, this form of responsive change is in contrast to a more doubtful, yet unfortunately, commonplace "pro-active" change, wherein change is viewed as inherently good simply by "shaking up" a complacent organisation (see [Mumford, 1995] for further discussion). Indeed, the early (and popular) works on BPR [Hammer, 1990; Hammer and Champy, 1993; Davenport, 1993] placed considerable emphasis on "fundamental and radical change" [Hammer and Champy, 1993] – tossing out the old and in with the new. Thus, it was asserted, "business reengineering isn't about fixing anything. Business reengineering means starting all over, starting from scratch." [ibid.] In later work, Hammer in particular claimed that early adopters of BPR "succeeded far beyond anyone's expectations" [Hammer, 1996]. Hammer's co-author, James Champy, however casts serious doubt on this claim, noting that many "shake-up" reengineering efforts have had less than sparkling success [Champy, 1995].

would-be collaborators to collaborate with us rather than with competitors. In other words, we cannot fully predict future business process change, since it is so heavily influenced by factors that are substantially beyond our control. That is, we need to expect the unexpected, and respond accordingly.

2.9. Exchange Theory

A branch of sociology termed Exchange Theory (e.g. [Cook, 1987b]) supports much of what has already been argued here. Exchange theory is built upon the realisation that collaborations (usually) result in some mutually satisfactory exchange from which all participants benefit. Hence, it is advantageous to each participant to collaborate. There are exceptions to this (absolute power holders [Molm, 1987], free riders, and zealots [Coleman, 1987] are good examples), but these are exceptional cases and will not concern us here. Rather, we shall focus upon the types of collaboration with which organisations tend to involve themselves. That is, collaborations structured around effectively balanced, and continually re-balanced, exchange relations.

2.10. Exchange Relations

In contrast to exchange theory, economists often treat collaborations as fundamentally market-driven, wherein organisations interact with and respond not to one another directly, but rather via a theoretical construct termed a market. A market is seen as an aggregation (a balancing) of the needs of all organisations and thus central to achieving "perfect competition" [Emerson, 1987]. Exchange theory counters that, although markets set the basis for some expectation of exchange rate and exchange behaviour, it is the establishment, responsive maintenance, and utilisation of individual exchange relations between individual parties that governs the actual process of exchange [Cook, 1987a]. An exchange relation can be seen as an established relationship between collaborators that is either strengthened or weakened by successive exchanges, according to the level of satisfaction experienced by each participant with the outcome of each exchange.

This is important, since it implies that we cannot expect to elicit impersonal and generalised business processes for given abstract markets. Rather, we need to be attuned to, and possibly deeply immersed in, the subtle and ongoing evolution of the highly personal exchange relations underlying the evolving business processes we intend to support.

2.11. Balancing Exchange Relations

Exchange theory tells us that one organisation[6] is dependent upon another if that second organisation controls some resource the other values and desires, and there are few, if any, other sources of supply [Yamagishi, 1987]. Exchange relations between collaborators are "balanced" if the organisations involved in exchange are equally dependent upon one another; otherwise an imbalance exists in the relation. Whenever there is a power imbalance, there are numerous power-balancing mechanisms that take place to restructure the exchange relation and restore equality [Cook, 1987a]. It is this balancing effect that causes exchange relations to endure, and that mandates much business process change.

2.12. Language/Action Perspective

To recap, business processes are fundamentally underpinned by, and change (often in unexpected ways) in support of, exchange relations. To understand how exchange relations are formed and utilised, we can turn to the Scandinavian Language/Action Perspective (LAP) [Goldkuhl et al. 1998b]. In a nutshell, LAP proposes *human communicative action* as fundamental to understanding, and thus explaining and supporting, organisational behaviour. In particular, we can think of LAP as proposing that organisations use language purposefully to nurture, and participate in long-term exchange relations. This perspective has a relatively long history, with its roots lying in Speech Act Theory [Austin, 1962; Searle, 1969], the Language-Game theory of Wittgenstein [Wittgenstein, 1958], the Theory of Communicative Action [Habermas, 1984], and the work of Winograd and Flores [Winograd and Flores, 1987]. LAP argues that organisational collaborations (i.e. exchange relations) are underpinned by communicative actions (usually supported by some material action - such as delivery of a physical product in exchange for payment). Communicative actions are *purposeful* speech acts, in the sense that they are performed specifically to induce their valued outcomes [Emerson, 1987] that contribute to goal satisfaction.

2.13. Business Action Theory

There are many branches of LAP, including Action Workflow [Medina-Mora et al. 1992], Business Action Theory [Goldkuhl, 1996], [Goldkuhl, 1998a], and DEMO [Dietz, 1994; van Reijswoud and van der Rijst, 1999]. All of these branches are bound

[6] Actually, exchange theory is not restricted to organisational behaviour. It has been noted that exchange theory is widely applicable to just about any social situation, since almost every social situation involves a mutual give-and-take [Blau, 1987]. Our focus here, though, is upon organisational exchange relations.

by the idea that collaborating parties use speech acts with purposeful intent to communicate needs, align goals, coordinate activities to satisfy those goals, and build enduring personal relationships (exchange relations) based upon mutual trust. This final point - the development of personal relationships based upon mutual trust (i.e. exchange relations) - is a particular emphasis of Business Action Theory.

2.14. BAT Transaction Pattern

Business Action Theory (BAT) [Goldkuhl, 1996; Goldkuhl, 1998a] argues that businesses use language to play a conventionalised *action game* (a universally understood, almost ritualistic, pattern of behaviour with multiple parties sharing and adhering to mutually recognised and accepted rules). That action game encompasses the pursuit, negotiation, and fulfilment of exchange agreements. The underlying philosophy is that organisations cooperate to arrive at mutual success and to establish long-term business relationships built upon mutual trust.

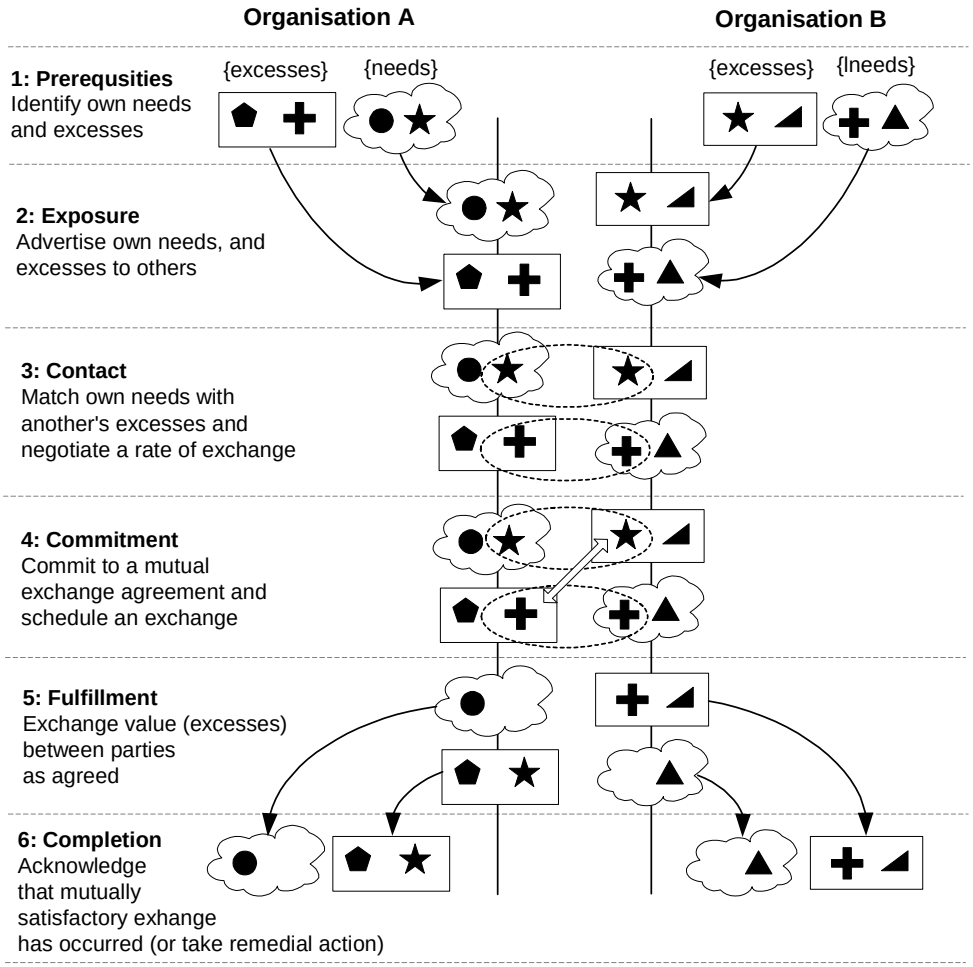

Figure 2.1 – BAT Transaction Pattern

BAT effectively takes exchange theory one step further, explaining not just that organisations embrace exchange relations, but also how organisations seek, form,

utilise, and strengthen them. More specifically, BAT perceives business interactions as pursuing a course described by the multi-phase transaction framework, or pattern, depicted in Figure 2.1, above.

In Figure 2.1, rectangles contain an organisation's excesses, and clouds contain their needs (or lacks). The various symbols within the rectangles and clouds depict abstractly the specific excesses and needs of the particular organisations shown. Matching excesses and needs (represented by dotted ellipses) indicate potential for exchange. Double-headed arrows depict an agreed exchange of various respective excesses in partial fulfilment of respective needs. Note that at the completion of an exchange, some needs may remain unfulfilled, and hence may be exposed again later, along with available excesses, for subsequent exchange.

The six phases of this BAT transaction pattern imply an ongoing interaction between the collaborating parties, wherein there is a continual process of exchange between them. This process involves exchanges of knowledge, interests, proposals, counter-proposals, commitments, and so on, but the eventual aim (the aim of the whole transaction) is a mutually satisfactory exchange of value (perhaps money for products, but possibly something less tangible like goodwill). During these continual exchanges, then, there is a continual negotiation (both explicit and implicit), throughout which each party is shifting their position in response to their collaborator's evolving communicative actions. Many of these necessary positional shifts may have been anticipated (price reduction, for example), but others will have been unanticipated (demand for payment via multiple currencies including the Euro, or provision of online support services over the Internet, being extreme examples). A succession of such unanticipated shifts over a series of exchanges constitutes the unanticipated business process change that is inherent in remaining commercially competitive.

2.15. Transaction History

We have seen in the BAT transaction pattern that the process of negotiation between the parties is aimed at determining a mutually acceptable exchange rate - aligning their perception of the relative values of their respective excesses - and a mutually acceptable schedule for exchange. The mutually satisfactory fulfilment of a negotiated exchange establishes trust and thus strengthens the exchange relation between the parties. Underpinning that strengthened exchange relation is the history of transactions between the parties, including the history of negotiated rates of value exchange and negotiated sequences of exchange, forming customary practice between them. It is these

historically constructed customary practices that are the basis of ongoing business relationships.

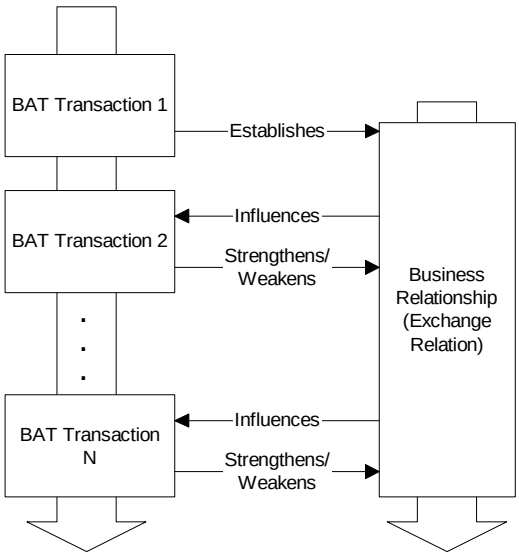

Figure 2.2 – Exchange Histories Cultivate Exchange Relations

2.16. The Human Touch

Interestingly, there is compelling evidence that businesses frequently view negotiated exchange agreements not as legal contracts (even in cases where they are clearly legally enforceable) but rather as "gentlemen's agreements" dependent more upon a tacit code of honour than any recourse to law [Macaulay, 1992]. There is, it appears, an underlying recognition that it is the "human touch" and the establishment of long-term personal relationships (exchange relations), rather than red tape, contracts, and lawyers that oils the wheels of industry [ibid.]. As an illustrative example, a number of senior developers and managers left our industrial collaborator (EDP) a number of years ago to form a start-up company (named Selima). Through a series of satisfactory exchanges, they steadily built good relationships with a group of long-term customers. Unfortunately, one customer, a furniture supplier, began to experience cash flow problems and was unable to pay for quite considerable goods and services already received from Selima. Rather than jeopardise their good relationship by taking this customer to court, negotiations revealed that the furniture supplier was currently stock-rich but cash-poor. Consequently, conventional payment was waived, and in return Selima was furnished with brand new desks and chairs throughout.

2.17. Review

This chapter has explored what business processes are and why they change. In summary, we have seen that organisations exist to satisfy their goals, and that goal

satisfaction frequently involves value-oriented inter-organisational collaboration. Collaborations that are mutually satisfactory to all participants lead to the formation of enduring long-term relationships, termed exchange relations, which subsequent collaborations can utilise and enrich. It is the seeking, forming, and maintaining of exchange relations, and their utilisation in ongoing collaborations that underpins an organisation's goal-contributing business processes. Now, as exchange relations evolve, business processes evolve accordingly, and often in unanticipated ways. This, of course, requires a corresponding adaptation of the information systems supporting them. An information system that fails to support evolving business processes threatens to undermine exchange relations, and hence potentially impedes the realisation of organisational goals. Such systems are termed *legacy systems*. They can hold a business back, and they are the focus of the next chapter.

Chapter 3

Legacy Systems

Has this fellow no feeling of his business?

- William Shakespeare, *Hamlet* V, 1

3.1. Preview

In the previous chapter we saw that organisations enact business processes to achieve their goals. Furthermore, organisations continually adapt their business processes (often in unanticipated ways) in order to remain competitive. Organisations develop and maintain information systems to support these changing business processes. Our concern in this chapter is information systems that fail to support business processes in the light of such ongoing business process change. As we saw in chapter 1, information systems that resist business process change are termed legacy systems, and to remain competitive organisations must tackle the legacy systems that hold them back. The answer, though, is not simply to migrate legacy systems to new technology; we shall see that therein lies the road to literal migration, wherein the old legacy system is migrated to a new legacy system, albeit one that is one hundred percent buzzword compliant. The real challenge with legacy systems is to uncover the underlying causes of their resistance to business process change (particularly unanticipated change), then respond to those causes with appropriate actions that eliminate their change resistance, and finally ensure that the resulting adaptable systems do not themselves eventually degrade back into legacy systems in the future.

In this chapter, we survey numerous strategies for dealing with legacy systems, focusing particularly upon a strategy incremental migration to systems that remain adaptable in the light of business process change. We note that there is little effective methodological support for such migration, and thus set the scene for such methodological support as a particular contribution to knowledge of this thesis.

3.2. Legacy Systems

As mentioned already, organisations develop, adapt, and deploy computerised information systems (as opposed to, say, real-time embedded systems) to support the enactment of their business processes. The requirements for an information system, then, are in the main derived from the business processes it supports. Since the job of an

information system is to support business processes, an information system only retains its value when it can be evolved in step with changes to the business processes it is intended to support.

Drawing inspiration from [Brodie and Stonebraker, 1995][7], we can define a legacy information system (hereinafter referred to simply as a *legacy system*) as an information system that significantly and continually resists the ongoing reflection of the evolving business processes it is intended to support. To be more precise, we can say that an information system is a legacy system if the business processes it is intended to support are changing at a rate that exceeds the rate at which the information system can itself be adapted to reflect those changes (see Figure 3.1).

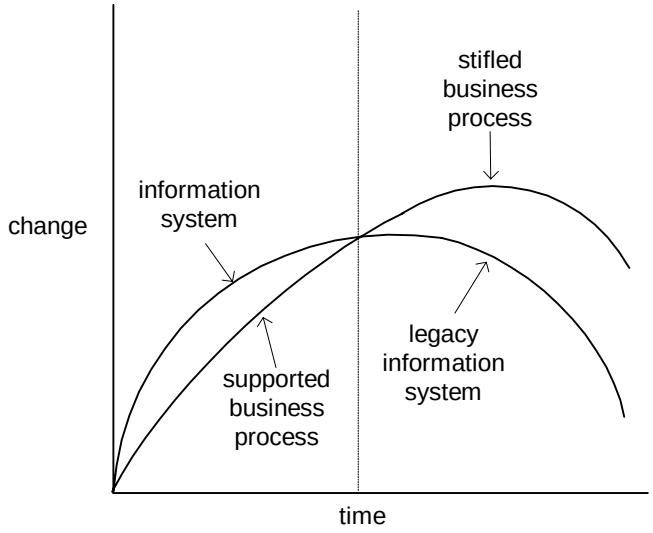

Figure 3.1 – Legacy System Falling Behind Business Process Change

A legacy system's resistance to the provision of adequate business process support in the light of ongoing change can stifle an organisation's ability to respond to competitive pressures and market opportunities and can, therefore, impede an organisation's commercial competitiveness[8]. Simply put, legacy (software) systems can hold a business back, so that the business itself becomes a (socio-technical) legacy system, unable to respond effectively to further change [Gold, 1998].

[7] Who themselves define a legacy information system as one that "significantly resists modification and evolution to meet new and constantly changing business requirements".

[8] Note that the prevalence of legacy systems confirms Manny Lehman's first law of software evolution [Lehman, 1980; Lehman, 1997], termed "the law of continuing change", which hypothesises that software in-use in a real-world environment must continually change or become progressively less useful in that environment.

3.3. Tackling Legacy Systems

The detrimental effects of a legacy system will not simply "go away" if left alone. On the contrary, the inevitable progression of requirements change ensures that the quality of an unchecked legacy will diminish over time[9]. Thus, an organisation faced with legacy systems has to make an active commitment to how it is going to deal with them. Several strategies are possible, and each has merits in certain circumstances. Choosing the appropriate strategy is vital, since, as the Software Engineering Institute has discovered, the number one reason "for reengineering project failure … [is where] the organization adopts a flawed or incomplete reengineering strategy" [Bergey et al. 1999]. Seven potential strategies are discussed below; others almost certainly exist. Note that it may be possible to combine some of these strategies, where appropriate. Each individual organisation must evaluate their own legacy systems, their level of change resistance, and the capabilities of the organisation itself [Brown et al. 1996] when determining which strategies best suit their particular circumstances.

Strategy 1: Maintain It: The simplest strategy is to change nothing at all. That is, to keep the legacy system, continue to maintain it in the same way, and simply accept that it will progressively fail to meet the needs of the business. This strategy may be appropriate for an organisation whose culture considerably resists change, and for whom change may prove more costly (financially, emotionally, intellectually) than the status quo. If the pain of tackling the legacy system is greater to the organisation than the pain of change resistance in the legacy system, then the status quo may prove appropriate. This is not all bad news; organisations are highly inventive, and can often work around the inadequacies of legacy systems by, for example, adopting manual procedures to circumvent legacy system shortcomings. However, this approach may simply be delaying the problem; ongoing requirements changes will only worsen the problem until a point it reached at which the pain of maintenance is so unbearable that one of the other available strategies may prove ultimately necessary.

Strategy 2: Freeze It: When maintenance of a legacy system drains more from the organisation than it could possibly return, it may be appropriate to freeze the legacy system and halt further maintenance work. Maybe the system isn't perfect, but maybe the business can live with its limitations. This may be a particularly appropriate strategy where the legacy system is not core to the business, and where the resources being

[9] That is, if we accept Crosby's succinct definition of quality as "conformance to requirements" [Crosby, 1979].

pumped into its maintenance would deliver greater value if they were diverted elsewhere.

Strategy 3: Outsource It: When software development is not the core competence of the business, it may be appropriate for the business to focus upon its core competence and hand responsibility for the legacy system to an outside organisation. A specialist software development organisation will almost certainly be more experienced in and better equipped for dealing with legacy system maintenance, and will free the owning organisation from the burdensome distraction of having to deal with the maintenance effort directly.

Strategy 4: Shelve It: There are occasions where the legacy system is more trouble than it is worth, and the business can put it on the shelf and forget about it. This is typically appropriate where the system is so far out of line with the needs of the business that it delivers almost no value whatsoever. In place of the legacy system, the organisation can either resort to a manual system, completely reengineer the process that required the system, or replace the system as described in the next strategy.

Strategy 5: Replace It: There may be commercial products available which could replace the legacy system. Many organisations have found the burden of maintaining (possibly several conflicting) legacy systems so troublesome that they have committed to some externally developed product, even where this involves some adaptation of their existing business processes to do so. Where appropriate, running the replacement system in parallel with the legacy system for a period may help phase the replacement in.

Strategy 6: Rewrite It: In the light of their inherent un-maintainability, it is often tempting to throw away a legacy system and write a replacement system completely from scratch. Getting rid of the problem "once and for all" is clearly appealing. Unfortunately, in most cases, a wholesale rewrite is unlikely to succeed. Later in this chapter we explore why wholesale rewrites are inherently risky and hence are less appealing than they first appear.

Strategy 7: Migrate It: As we shall see, the most difficult strategy of all may be to grasp the nettle and migrate the legacy system piecemeal to a replacement system that does support the business successfully. Migration, in this context, is essentially synonymous with reengineering. The generally accepted definition of reengineering is “the examination and alteration of a … system to reconstitute it in a new form” [Chikofsky and Cross II, 1990], although with migration we add that the work is best

achieved incrementally, and the purpose of that effort is to enhance support for continually changing business processes. It was pleasing, therefore, to come across the following definition for reengineering late in the preparation of this thesis: “Reengineering offers an approach to migrating a legacy system towards an evolvable system” [Reengineering Center, 1995]. Migration is appropriate where the legacy system does support the business in many ways, and where its shortcomings are (at least eventually) recognised and understood. The aim is to keep what is good about the legacy system whilst simultaneously eradicating the resistance to change that prevents it from continuing to support the business effectively. The most compelling reason for adopting this strategy can be summed up in the phrase "better the devil you know, than the devil you don't". Migration allows us to build upon our successes and use this as leverage with which to tackle our failures. Although migration is probably the most demanding of all seven strategies (requiring us to “grab the bull by the horns”), we shall see it is also potentially the most rewarding. Consequently, it is upon this strategy that much of this thesis concentrates.

3.4. Risky Rewrites

Perhaps the most tempting of the strategies outlined above is to simply start all over again, completely rewriting the legacy system from scratch (see strategy 6 above). This wholesale approach has been variously referred to as replacement [Warren, 1999], big bang [Weiderman et al. 1997], and cold turkey [Brodie and Stonebraker, 1995]. It is this approach to which EDP (the industrial collaborator on this research) has historically been drawn. Unfortunately, as EDP has found, although wholesale rewrites sound tempting, they bring with them substantial risks of failure.

Brodie and Stonebraker highlight ten significant risks associated with wholesale rewrites [Brodie and Stonebraker, 1995], which can be summarised briefly as follows: A wholesale rewrite is a mammoth effort which will be hard to envision, hard to manage and hard to complete. Management is unlikely to offer budgetary approval for such a mammoth effort unless additional business functionality can be promised, which would add complexity to the project and increase risk of failure. There is likely to be considerable resistance towards the project and hence lack of cooperation throughout the organisation due to fear, doubt, and impatience. Even if the replacement system ever gets off the ground, it is likely to be missing features, behind schedule, and hard to cut over to. All in all, a one-shot rewrite is difficult to get going, and highly likely to either fail or be terminated.

3.5. Incremental Migration

Given the dangers of wholesale rewrites, we may be better served by taking the existing legacy system as a starting point from which to begin construction of a replacement system. This calls not for a one-shot approach but for a process of *incremental migration*. By incremental migration we mean that we address the migration one piece at a time. By addressing legacy systems incrementally, we reduce the risks inherent in an all-or-nothing rewrite. In particular, at the end of each increment we are able to test that we have done no harm, with the potential to rollback to a previous "good" increment should this prove necessary.

Brodie and Stonebraker refer to incremental migration as Chicken Little since (in honour of the children's book character of the same name) it prevents the sky from falling in [Brodie and Stonebraker, 1995]. They highlight several advantages of an incremental approach over one-shot rewrites: The legacy system can be kept running throughout incremental migration. Since there is a gradual phase-in of changes, rather than a dramatic switch from old to new systems, risks are reduced and users and developers are able to adjust to the changes over time. Finally, this risk reduction increases the likelihood of management buy-in since it leverages and protects their current legacy investment.

3.6. Software Ageism and Techno-Centric Migration

Unfortunately, even when adopting an incremental migration approach to tackling legacy systems, we are still susceptible to the danger that the replacement system will itself be a legacy system. This danger is particularly acute when we place our primary focus upon the differences between the implementation technology of the existing legacy system and that of the envisioned replacement. This is a surprisingly common trap. Indeed, many well-respected authors define legacy systems not in terms of change resistance but in terms of both age and implementation technology. For example, "The term 'legacy system' describes an old system … typically developed according to dated practices and technology" [Warren, 1999]. Similarly, the "Three Amigos" give the following definition: "Usually an old system that is created using more or less obsolete implementation technologies." [Jacobson et al. 1999]. As a final example, "[L]egacy systems … [are] usually more than seven years old [!] … and entail outmoded or varied proprietary technologies" [Miller, 1997].

The problem with such definitions is that their focus on age and implementation technology completely misses the point that the real issue with legacy systems is their

inherent resistance to reflecting ongoing business process change. Even Brodie and Stonebraker [Brodie and Stonebraker, 1995], whose definition initially inspired our own, quickly drift away from concerns of tracking business process change and instead focus their book almost wholly upon mechanisms for porting from one technical platform to another.

3.7. Addressing Change Resistance

The problem is that there still remains a further major risk that Brodie and Stonebraker do not mention in their ten risks of wholesale rewrites (see above), and that would not be addressed simply by adopting an incremental approach. That risk is that unless we are clear about the underlying causes of change resistance within a given legacy system, and purposefully act to eliminate them and prevent them from recurring, we are in real danger of the replacement system perpetuating them. The replacement system may well end up being the result of an all too literal migration from an old legacy system to a new legacy system, albeit one cast in modern technology.

Simply relying on novel and popular technologies, then, is not the answer to successful migration; there is, for example, already a history of object-oriented legacy systems [FAMOOS, 1999]. Although age and implementation technology concerns are important, focusing upon them as the central issue when dealing with legacy systems is likely to have only marginal impact upon the ability of a replacement system to reflect ongoing business process change.

By focusing on the implementation technology of the replacement system rather than on the change resistance of the legacy system it is replacing we are in danger of missing the lessons the legacy system can reveal to us. If we don't learn from the past, we are doomed to repeat it. A legacy system has important lessons to teach us about where we went wrong. We need to listen to the legacy system, learn from our mistakes, and protect ourselves from repeating them.

3.8. Adaptable Systems

If a legacy system is a system that resists evolution in response to business process change, then a non-legacy system (an *adaptable system*) is not simply a web-enabled, object-oriented, distributed, component-based, buzzword-compliant killer application. Rather, an adaptable system is a system that *can* evolve in step with and in response to evolution in its requirements, reflecting ongoing change in the business processes that the information system is supporting. Contrast Figure 3.1 above with Figure 3.2 below:

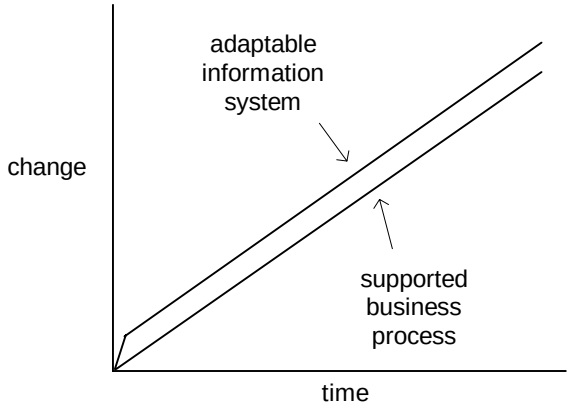

Figure 3.2 – Adaptable System Tracking Business Process Change

Adaptable systems allow us to evolve the business according to business needs rather than the (limiting) capacities of our legacy systems, thus avoiding what has been termed "technological determinism" [Ramage et al. 2000a] in the shaping of the business.

To address legacy systems effectively, and realise adaptable systems with which to replace them, we must focus not upon implementation technology alone, but rather upon the underlying root causes of each specific legacy system's resistance to requirements change. Once we understand why a particular legacy system resists the reflection of requirements change, we are better positioned to tailor our migration strategy to directly address that resistance. Surprisingly, we have been unable to uncover in the literature any effective methodological support for such migration. This thesis aims to fill that gap.

Once a legacy system has been migrated to an adaptable system, the challenge then shifts to the adoption of a perpetual process of *monitoring the evolution* of the adaptable system with an eye to nipping further change resistance in the bud. The result is a system that is continually receptive to evolution in step with the business processes it supports. In other words, not only do we need to deploy a *migration* process, we also need to adopt a perpetual process of adaptable systems *evolution* to prevent the adaptable systems to which we migrate from ever again degrading into change resistant legacy systems.

3.9. Review

We have seen that a legacy system is an information system whose requirements (i.e. supported business processes) change at a faster pace than the potential rate of evolution of the information system itself. Tackling a legacy system is a matter of closing that gap. To understand how that gap formed in the first place, we need to understand how once adaptable systems degrade over time into legacy systems. By understanding the

mistakes of the past, we can learn how to avoid them in the future. When we better understand why and how legacy systems become change resistant, we will be better positioned to address that change resistance in existing legacy systems, and to ensure that newly adaptable systems do not themselves suffer from eventual degradation into legacy systems. In the next chapter we examine how a commonplace strategy for accommodating requirements change results, paradoxically, in an inability to respond to further requirements change. In later chapters we reveal a novel and adaptable legacy system migration strategy which, when supported by personalised catalogues of migration expertise, reverses the effects of such change resistance, resulting in adaptable systems, responsive to ongoing business process change.

Chapter 4

Petrification

Our doubts are traitors,
And often make us lose the good we might win,
By fearing to attempt.
- William Shakespeare, *Measure for Measure* I, 4

4.1. Preview

In the preceding chapter we saw that legacy systems resist the reflection of ongoing business process change. In this chapter we shall see that an information system may actually begin its life with a purposefully *adaptable architecture* responsive to (at least certain anticipated) requirements changes. Unfortunately our desire to control risk frequently leads paradoxically to repeated waves of *hacking* that tend to *petrify* mature information systems resulting in the change-resistant *accidental architectures* that underlie legacy systems. Such architectures emerge more from the ingenuity of the systems developers' ability to "warp" the code than from any purposeful design. As successive waves of hacking mutate an information system further and further, its accidental architecture becomes more and more convoluted, and the intellectual capability of developers to identify and apply clever hacks diminishes to a point at which the system is declared un-maintainable.

4.2. Disgraceful Maturation

A naïve assumption about legacy systems is that they were badly designed in the first place. Although there may be some truth to this in extreme cases[10], it is unusual for professional software developers to create change-resistant software wilfully. On the contrary, as we shall see in chapter 6, most popular software development methodologies generally strive to develop new information systems architected to be, at the very least, receptive to certain forms of *anticipated* change.

[10] For example, as pointed out in chapter 1, change resistance was essentially unavoidable during the earliest days of computing, when we were stifled by severe hardware limitations. In addition, early development efforts did not have the benefit of modern-day development practices [Osborne and Chikofsky, 1990], and developers were probably unaware of many of the maintenance problems they were causing and how they could be best avoided. Furthermore, early assumptions about the potential lifespan of software systems (see chapter 1) certainly won't have helped.

Unfortunately, and herein lies a major contributor[11] to the prevalence of legacy systems, such methodologies also tend to result in information systems that mature *dis*gracefully in response to *unanticipated* requirements change. As we shall see, the belief that all future changes can be anticipated encourages successive waves of mismanaged evolution, in a practice often termed *hacking* (described below). Repeated hacking leads to tangled software architectures, which begin to "creak under their own weight" so that it becomes progressively harder and harder to adapt them to reflect any changes in their requirements (either anticipated or unanticipated). To describe this process, we introduce the term *petrification*, wherein a once flexible information system architecture solidifies over time so that it resists the reflection of further ongoing requirements change[12].

4.3. Inflexibility and Brittleness

The term *petrification* was chosen carefully, since it has a dual denotation; both a solidification of systems' architectures, and a paralysing fear amongst developers unwilling to make changes to legacy systems in case unanticipated and damaging side effects result.

Solidification of a system's architecture makes it difficult to enact changes to that system due to the sheer effort involved. This is often termed system *inflexibility* ([Brodie and Stonebraker, 1995; Weiderman et al. 1997]). Developers' fear of making changes, on the other hand, stems from system *brittleness* [ibid.] wherein even relatively small changes can cause the legacy system to break apart. When responding to petrification, then, we need to encourage software development practices that both make systems more flexible and increase the confidence of developers that they will do no harm when responding to ongoing requirements change.

In other words, it now seems clear that it is rarely the case that a legacy system cannot be adapted to reflect requirements change, but rather that such work tends to be both difficult and unnerving. The aim of this thesis is to tackle legacy systems by appealing

[11] Certainly, however, not the only contributor; legacy systems are clearly the result of many complex socio-technical interactions. As succinctly put by [Mitleton-Kelly and Papaefthimiou, 2001] "[legacy systems] emerge from the intricate interrelationships of diverse elements related to business, market, organisational and technological aspects that are part of the social ecosystem within which … organisation operates". Nonetheless, within the context of the actual legacy systems themselves, we have found the specific socio-technical concerns surrounding the system development methodology to have a major impact on architectural adaptability (which we see as central to absorbing requirements change).

[12] The prevalence of petrification in legacy systems confirms Lehman's second law of software evolution [Lehman, 1980; Lehman, 1997], termed *the law of increasing complexity*, which hypothesises that as a program changes, it structure degrades, and its complexity increases, unless deliberate steps are taken in response to this.

to software development practices that make system adaptation both simpler and less terrifying.

4.4. Software Architecture

So far in this chapter we have used the term *architecture* several times without providing a definition. Although we intuitively accept that software architectures do exist, the term has actually proven quite hard to define. Perhaps the most widely used definition is that of Shaw and Garlan: "The architecture of a software system defines that system in terms of computational components and interactions among those components." [Shaw and Garlan, 1996]. Definitions like this one essentially view software architectures as structures, which indeed they are. Unfortunately, such definitions fail to point out that just because something has an architecture does not mean that it was actually architected! Missing here, it seems, is a distinction between what we may term *legitimate* and *accidental* architecture. Legitimate architectures are architectures that are deliberately constructed with a specific purpose in mind, whereas accidental architectures are a coincidental (i.e. accidental) side effect of the construction process.

4.5. Accidental Architecture

One outcome of this thesis is the realisation that legacy systems (at least those we have studied) tend to be saddled with essentially *accidental architectures*, whereas the adaptable systems we have studied tend to boast *legitimate architectures* conceived with particular forms of adaptability in mind. In the rest of this chapter we shall see that when the set of adaptability assumptions reflected in a legitimate architecture are circumvented, without appropriate and deliberate *rearchitecting* (defined below) in response, the architecture starts to become an accidental by-product of adaptation and the slip towards petrification begins.

4.6. Legitimate Architecture

A legitimate architecture encapsulates certain assumptions about potential future change. That is, it reflects a set of *commonality* and *variability* assumptions (to borrow terms from the product-line community, e.g. [Weiss and Lai, 1999]). By this we mean that a legitimate architecture is deliberately constructed to accommodate certain forms of anticipated change (variability) around a framework that remains essentially stable (the commonality). In a legacy system, those assumptions proved to be insufficient in the long term and were circumvented in undisciplined and damaging ways.

Commonality and variability assumptions stem from a prior understanding of the domain being supported by an information system. Frequently, there is a process of *domain engineering*, which has been defined as "a software design discipline that focuses on the abstractions of a business (a *domain*) with the intent of reusing designs and artifacts" [Coplien, 1998b]. Domain engineering comprises *domain analysis* intertwined with *application engineering*. Domain analysis [Prieto-Dias and Arango, 1991] uncovers and captures an understanding of the domain of interest in terms of what tends to remain stable (points of commonality) and what is prone to change (points of variability) across that domain. Application engineering [Peterson and Stanley, 1994] constructs legitimate software architectures reflecting these commonality and variability assumptions.

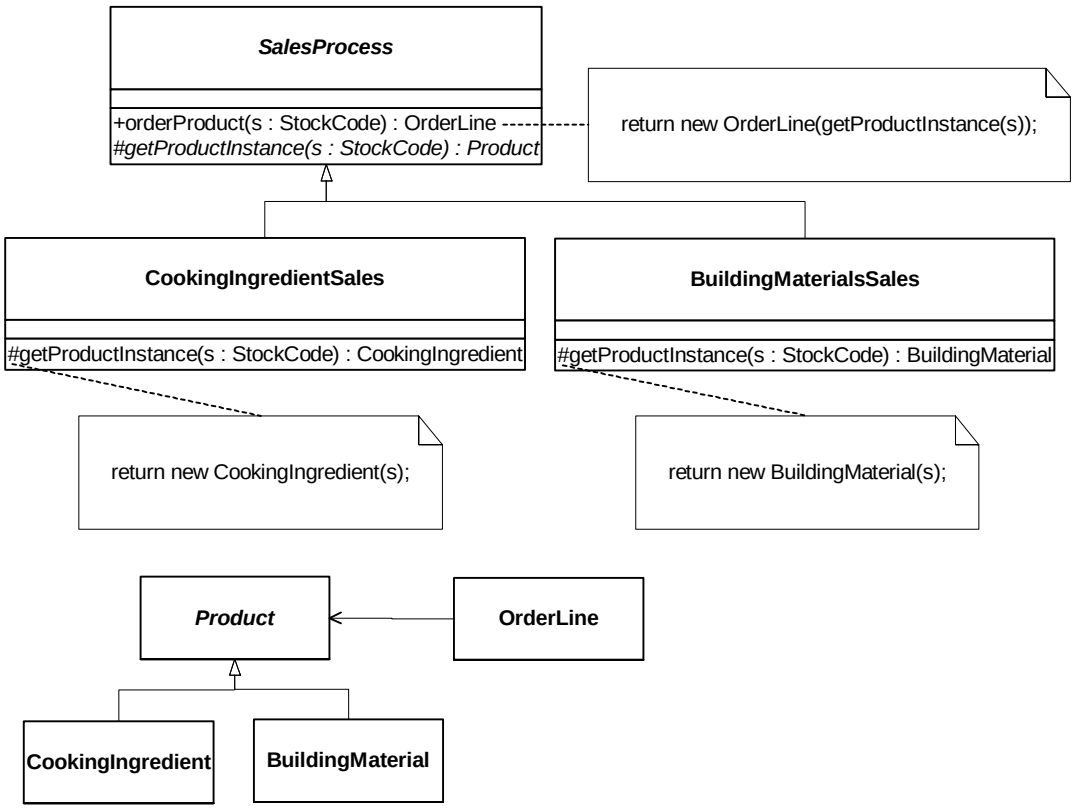

Figure 4.1 – Legitimate Architecture

Figure 4.1, above, expressed in UML[13] and adapted from [Lauder and Kent, 2000c], depicts a legitimate (initial) architectural fragment, based on an information system at EDP, to support (at least partially) the sale and distribution of both building materials and cooking ingredients. There is much similarity in the processes for buying these two classes of product, such as the need to create order lines, and the ability to get an

[13] The UML notation [Rumbaugh et al. 1999] has been used for many of the diagrams throughout this thesis, simply because it is a well established and prominent notation with which we are intimately familiar.

instance of a product. These points of commonality are reflected in Figure 4.1 via the abstract class SalesProcess, from which CookingIngredientSales and BuildingMaterialsSales are derived, and also an abstract Product class from which CookingIngredient and BuildingMaterial are derived. Although much detail is deliberately omitted from Figure 4.1, we can see that every OrderLine is associated with a Product (of one of the two derived types), and every class derived from SalesProcess is required to return an instance of a class derived from Product when given a stock code.

In addition to these commonalities, there are, of couse, differences between the two subclasses of SalesProcess, and also between the two Product subclasses. Thus, the getProductInstance method of CookingIngredientSales returns a new instance of CookingIngredient, whereas BuildingMaterialSales returns a new instance of BuildingMaterial. Likewise, although this is not depicted in Figure 4.1 for reasons of brevity, cooking ingredients tend to be perishable, and hence each instance of CookingIngredient would record a sell-by date. The short sales-window for CookingIngredients will also mandate fine judgments with regard to both stock ordering and incentive offers. BuildingMaterials, on the other hand, tend to be non-perishable, and thus may enjoy a longer window of opportunity for sales.

Figure 4.1, then, captures, as explicit points of variability[14] against a backdrop of commonality, (at least part of) our current understanding of the differences between the business process requirements of the two sales domains.

We will return to this particular example later in the chapter, when we respond to newly emerging requirements without adequately re-evaluating our commonality and variability assumptions, and hence upset our legitimate architecture through a series of (ultimately misguided) “hacks”.

4.7. Reuse Across Space and Time

Legitimate architectures are constructed with reuse in mind. Reuse can be across two orthogonal dimensions: *reuse across space* and *reuse across time*. By reuse across space we mean that we can deploy a given architecture across a *broad range* of applications at a given time, each adapted to satisfy the needs of a particular band of customers. By reuse across time, on the other hand, we create *multiple generations* (or *versions*) of a given application one following on from another. Reuse across time enables us to offer

[14] Since we are using the UML notation here, we depict variability via the object-oriented concepts of inheritance and covariant return types.

long-living products, providing evolution of an application in response to changing customer needs.

4.8. Anticipated vs. Unanticipated Change

Unfortunately, domain engineering results in architectures whose commonality assumptions are valid across space, but not necessarily across time. In essence, a domain analysis is born from a snapshot of the problem domain of interest, giving a comprehensive picture of how things are at a given moment. From domain analysis we can then construct (legitimate) architectures that accommodate a wide range of well-grounded solutions adapted at anticipated points of variability (hot spots [Pree, 1994], plug points [Szyperski, 1998], abstract classes [Woolf, 2000], etc) to accommodate diverse *current* customer needs.

Reuse over time, on the other hand, is a different matter. Legitimate architecture can only be grounded in *anticipated* change. Whereas domain analysis can reveal anticipated changes across space, it cannot accurately predict changes across time[15], since, as we saw in earlier chapters, "It is inherently impossible to determine in advance which direction change will take" [Mitleton-Kelly, 2000a]. Consequently, commonality and variability assumptions encapsulated in a given architecture may not hold for future generations of that architecture.

It is our response to the potential of *unanticipated* change to undermine a system's architectural commonality and variability assumptions that can lead to petrification.

4.9. Built-In Flexibility via Late Binding

To cope with unanticipated change, several researchers have attempted to make their systems extremely flexible by delaying some architectural decisions until run-time. For example, we can use reflection wherein a program is allowed at run-time to make changes to its own meta-level, as proposed by both [Kiczales et al. 1991] and [Foote and Yoder, 1996], among others.

As another example, we can rely on *dynamic systems* [Henderson, 1998], which delay component composition until run-time. Examples of dynamic systems approaches are FLEXX [Rank et al. 2000], which adopts a pipes-and-filters architectural style with late

[15] Indeed, it has been argued that for any given domain, although there will probably exist a small number of primitive "domain atoms" intrinsic to the domain (such as price, currency, bond, contract, etc. for banking), and which are relatively immutable over time, we have to assume that "*everything* else – the technology beneath and the business above – is subject to change." [Millea et al. 2001].

re-composition permitted via a graphical tool[16], and our own EventFlow (nee EventPorts) model [Lauder and Kent, 1999a], which adopts an implicit-invocation architectural style, permitting run-time substitution of loosely coupled components.

Such late-binding approaches are appealing in the sense that they can protect us from (some) premature architectural commitments. However, they do tend to incur a performance penalty, which in many domains makes them practical only for prototyping. Thus, the "real" system in production would generally not benefit from their run-time flexibility. To resolve this problem, late-binding approaches are often restricted to high-granularity components, so that the majority of smaller-grained (internal) components cannot take advantage of the flexibility advantages of late binding.

An additional and greater problem, though, is that late binding approaches still require a commitment to an essentially stable commonality and variability assumption set, albeit with run-time binding at variability points. Thus, reflective systems tend to be reflective only about certain aspects: we must decide up-front which aspects require meta-level adaptability and this can only come from a snapshot of the domain being supported.

Similarly, dynamic systems approaches generally depend upon the existence of component types (although not implementations) with predetermined connection points. Commitment to which types of components are permitted, and the interface (or message) requirements of their pre-defined connectors, is an up-front decision on domain commonality. Dynamic binding (plugging) of component implementations at the available connectors exploits pre-determined points of variability.

In other words, run-time flexibility can only protect us from unanticipated requirements change at predetermined points of (run-time) variability, in which case that requirements change cannot have been entirely unanticipated at all! The real problems occur when truly unanticipated requirements changes occur, i.e. at previously determined points of commonality. Built-in flexibility cannot help here; rather we still need to respond to unanticipated change by continually reframing our commonality and

[16] The most recently published material on FLEXX [Hong et al. 2001] indicates that research into a more elaborate approach has begun, which supplements pipes-and-filters with multi-level architectures, internal software bus based communication, and multi-threaded dependable "atomic actions". One potential danger here would seem to be that as continually more elaborate run-time flexibility is attempted, run-time configuration may become indistinguishable from programming (albeit in an essentially visual language), requiring considerable technical expertise on the user's part. Interestingly, [Edwards and Millea, 2001] advocates precisely this approach of user actioned programming (via domain specific languages) as a means of "avoiding the hard question 'what are the system requirements' which is only ever answered badly by fixing a moving target."

variability assumptions and re-architecting to embrace flexibility that we had not previously anticipated.

4.10. Maintenance and Evolution

When we respond to requirements change we can appeal to two forms of software *adaptation*. Changes to a system that maintain our commonality and variability assumption set may be termed software *maintenance*[17]. We define *evolution*, on the other hand, as a reformulation of a system's architecture to reflect new commonality and variability assumptions. Maintenance, then, *maintains* the current architecture whereas evolution *evolves* it. It is our assertion (elaborated below) that a legacy system resists evolution, and in severe cases may even resist maintenance (due to extreme brittleness). An adaptable system, on the other hand, is responsive to both forms of adaptation.

4.11. Risk Avoidance

In the short term, software evolution (i.e. rearchitecting) is inherently more risky than software maintenance (i.e. localised change). We have seen that legitimate architectures are derived from a deep understanding of the points of commonality and variability inherent in a given problem domain, and have been designed, crafted, and debugged over potentially many generations of a software product. Architectural evolution, therefore, must only be undertaken with a deep understanding of its impact upon the accurate reflection of the needs of the problem domain being supported. Mismanaged architectural evolution has the potential to negatively impact the whole of a software application. Localised maintenance, on the other hand, is usually far less risky in terms of its scope of impact. Localised changes at predetermined points of variability (i.e. at points that were *expected* to change) tend to have the potential for only limited "collateral damage" when mishandled. Consequently, risk avoidance tempts us to, wherever possible, avoid architectural change and localise change to individual points of predetermined variability. It is tempting, then, to strive to respond to requirements change via maintenance only.

[17] This deviates somewhat from many other definitions of maintenance (see [Pigoski, 1997] for a good survey), which tend to clump together all forms of system change under a single definition, and do not differentiate between those changes that evolve a system's architecture and those that perpetuate it. Having said that, [Swanson, 1976] does enumerate three sub-categories of maintenance: corrective ("bug fixing"), adaptive ("porting"), and perfective ("new requirements"). Although it is probably valuable to differentiate between these *causes* of system change, our major interest in this thesis is the *impact* of those changes on the system's ability to absorb future change; hence the slightly different slant of the definitions we use here.

4.12. Hacking

As a quick recap: Where there is uncertainty and fear about the risk of architectural change, developers will often work hard to localise all change to established variability points. That is, developers will frequently be drawn towards maintenance, even where unanticipated requirements changes cry out for architectural evolution, force fitting localised changes that circumvent the commonality and variability assumptions of the software architecture. This activity of circumventing architectural assumptions via convoluted (and often radical) mutation of known variability points is known informally as *hacking*[18], and changes to an information system resulting from this are usually referred to as *hacks*.

We have found in our own development work that hacking is particularly prevalent when developers are working with code that they did not originally develop themselves, and hence with which they are not intimately familiar. The traditional separation of development groups from maintenance groups promotes this. However, we have also observed that even where developers are working with their own code, hacking remains a tempting strategy, due to the perceived inherent localisation of side effects.

4.13. Really Neat Hacks

Hacking is often highly skilled and creative work, delivering the maximum reward for the least possible effort. Often development mangers put their "best developers" on a requirements change that needs a "really neat hack" to force fit it into an architecture that was never intended to accommodate such change. The talents necessary for effective hacking are often held in high regard by professional programmers, and have even been celebrated in a popular book [Levy, 1994].

Below, we look at a few “neat hacks” in response to requirements change for the SalesProcess that was depicted in Figure 4.1 above. Note that none of the “hacks” we are about to see has been artificially invented. Although class names, and so on, have been changed slightly, the actual hacks themselves are all taken from real code changes made by well respected developers to real applications at EDP.

As we deploy our information system, we learn that there are actually considerable differences in the way builders and chefs want to order goods from us. For a start,

[18] There is, of course, another more damaging form of hacking, wherein ignorance of the present points of commonality and variability within an architecture can compel a developer to hack unnecessarily. This, however, is more a matter of education than architecture. In addition, there is a distinct and well-known use of the term hacking to imply wilful (and often criminal) undermining of system security (see, for example, [Taylor, 1999]) but this is not relevant to us here.

although we could probably loosely equate building bricks with sugar cubes, and bags of sand with flour, the actual way in which bricks and sands are bought and sold is fundamentally different from the way in which sugar and flour are bought and sold.

As an example, when ordering cooking ingredients, a chef may ask "What's fresh this morning?" or, when told that rosemary is out of stock, may ask, "What else goes well with swordfish?" If a particular meal proves popular, the chef may request "the same as last Thursday, but twice as much lamb". Thus, there is an essentially interactive process of negotiation between buyer and seller.

A builder, on the other hand, could not accept pebbles as a substitute for sand, and doesn't care that his timber was "cut fresh this morning". To a builder the product list being ordered is not negotiable: if the order cannot be fulfilled it will be taken elsewhere. The process, then, is one not of what is being ordered, but of how it will be delivered. Thus, if a particular depot is out of a certain type of window-frame, the process for the supplier becomes a logistics problem of finding another depot that can deliver that window-frame rather than of attempting to convince the buyer that skylights are a good alternative to windows.

The architectural fragment depicted in Figure 4.1 does not support this type of behaviour. However, if we are only personally involved in the development of this particular architectural fragment, we can not be sure of the assumptions other architectural fragments in the system make about the stability of this architectural fragment's publicly visible interface[19]. We do not know, for example, whether other architectural fragments assume that CookingIngredientSales and BuildingMaterialSales exist as subclasses of SalesProcess. Nor do we know what would happen if the orderProduct method returned null when a product is unavailable, requiring a call to a new findAlternateProduct method. All we do know is that externally visible changes to our architectural fragment have the potential to upset whatever assumptions external fragments may make, with any unanticipated (and perhaps subtle) side effects requiring (potentially substantial) post-delivery debugging to uncover and repair. This is particularly problematic if those external fragments have themselves been hacked, so that their underlying assumptions are fiendishly difficult to unravel even after close code inspection (see Implicit Architecture, below).

[19] We are using the term interface here more generally than the restrictive sense of function signatures, and include all externally visible properties.

To reduce the likelihood of unanticipated side effects we need to respect unknown external assumptions as far as possible. To achieve this we can "cleverly" preserve the eternally visible aspects of our architectural fragment by force-fitting the new requirement onto the side of the existing architecture, using a small "hack" as depicted in the architectural fragment of Figure 4.2.

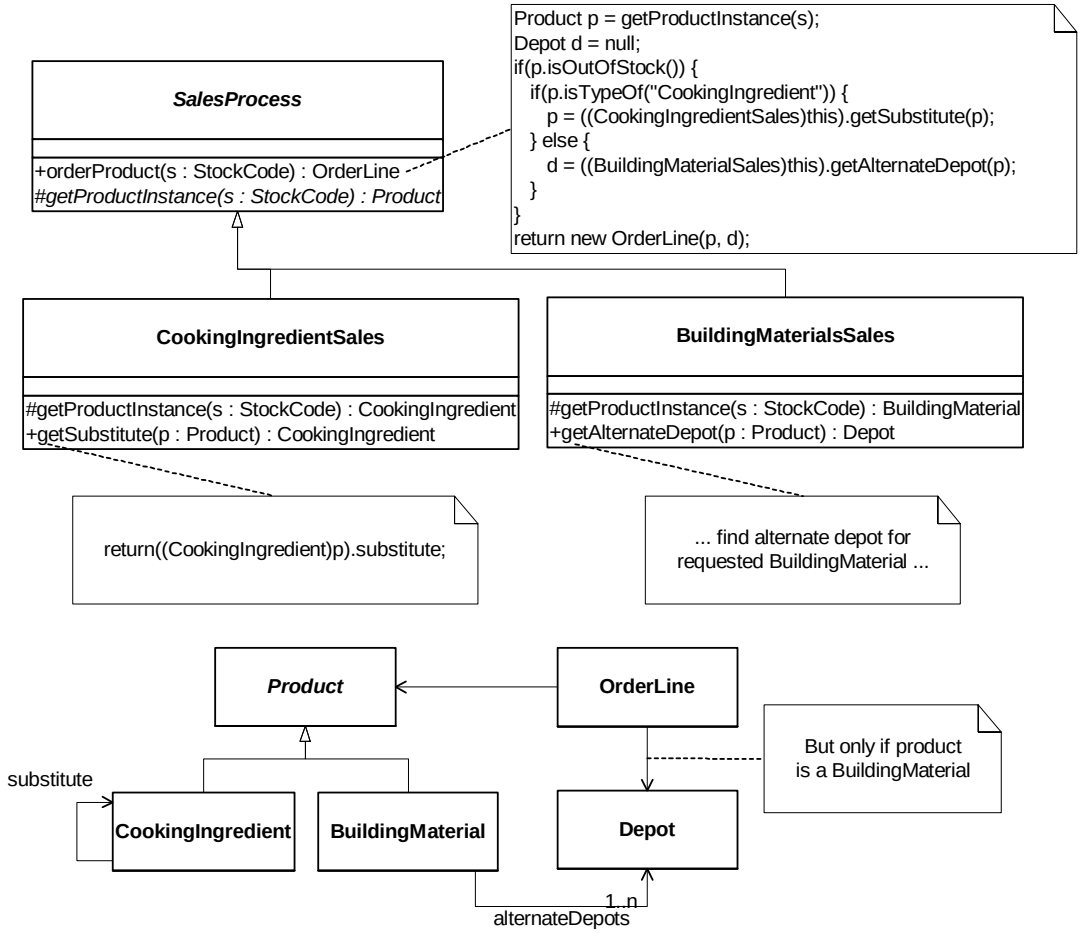

Figure 4.2 – Hack to Handle Out-Of-Stock Products

Note that in Figure 4.2, we haven't needed to change the architectural commonality and variability assumptions as such; rather, we have successfully circumvented them by adding a separate method to handle out-of-stock products in each of the CookingProductSales and BuildingProductSales classes. Of course, these separate methods (getSubstitute and getAlternateDepot) undermine the polymorphic behaviour of orderProduct somewhat, but a little run-time type checking in the orderProduct method soon sorts that out.

The business then decides to start selling KitchenWare (such as pots and pans). KitchenWare is interesting, since sometimes a substitute may be acceptable to the Chef (perhaps all thermometers are essentially equivalent to some chefs, for example),

whereas in other cases, the requested item must be found at any cost (e.g., a very specific carving knife is needed).

Now, a Product that has properties similar to both a CookingIngredient and a BuildingMaterial was rather unexpected. However, even though KitchenWare isn't really either a CookingIngredient or a BuildingMaterial, since it "behaves like" both of these existing product classes, we can implement the requirement quickly by some creative multiple inheritance, as shown in the architectural fragment in Figure 4.3, below. The same trick works for KitchenWareSales, which is "likeA" (although admittedly not "isA") CookingIngredientSales and BuildingMaterialSales. Of course, we then need to be able to tell which of these "modes" KitchenWareSales is operating in. We can accomplish this by a public data member in the KitchenWareSales class which external code has to remember to set before calling orderProduct.

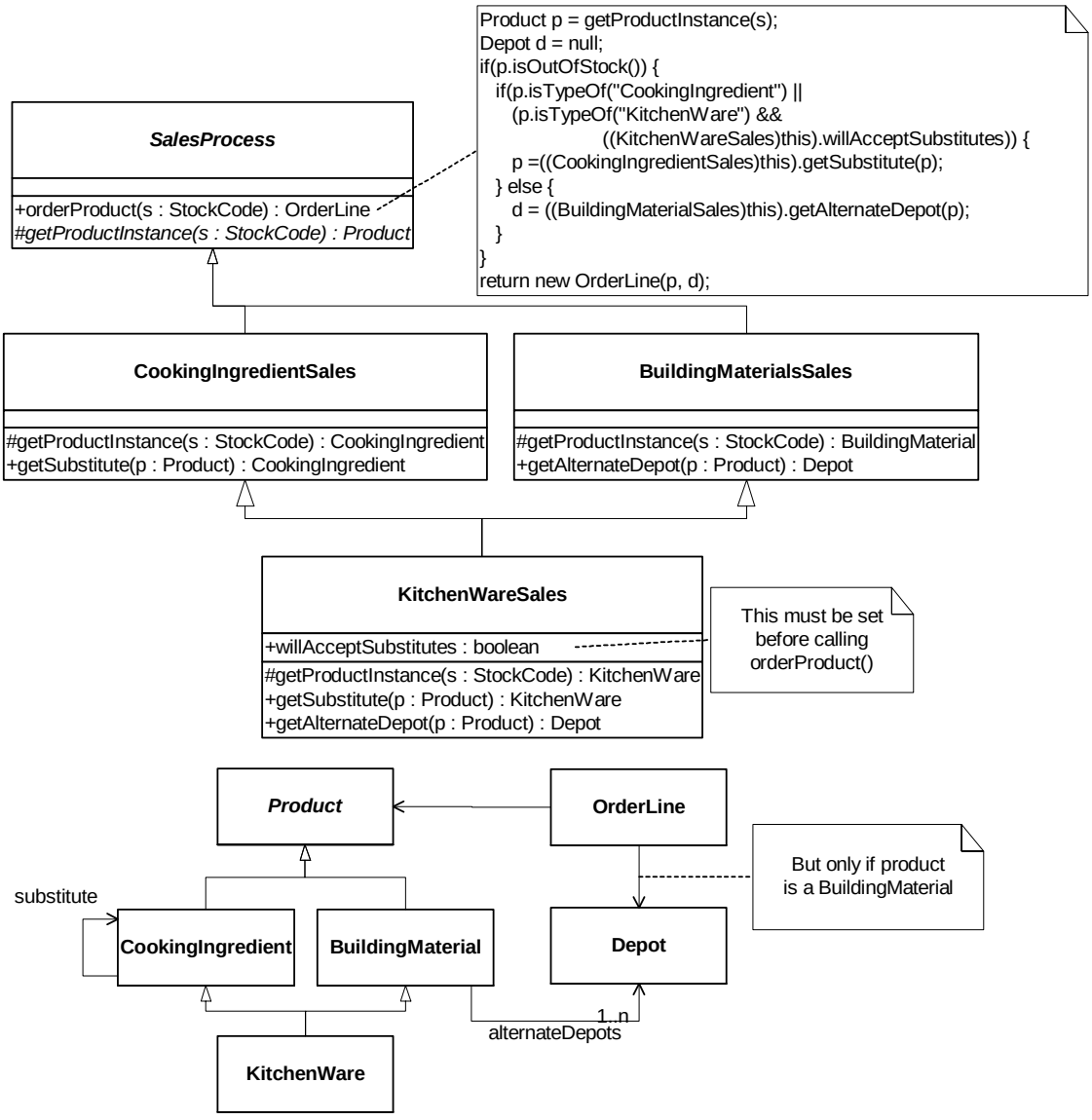

Figure 4.3 – Hack to Handle KitchenWare

In Figure 4.3, then, by applying a couple more clever hacks, we have again been able to circumvent (yet still maintain) the original architecture's commonality and variability assumptions. By localising changes we have reduced the likelihood of worrisome side effects on external code (dependent upon our original architecture) that could result from a (larger) architectural change.

4.14. Implicit Architecture

Hacking, like that depicted in Figures 4.2 and 4.3 above, may well be a compelling short-term strategy for risk avoidance. Unfortunately, a hack essentially layers an *implicit architecture* on top of the system's *legitimate architecture.* For example, Figure 4.3 shows that an OrderLine may refer to an alternate Depot only if the Product is a BuildingMaterial. Since KitchenWare is a subclass of BuildingMaterial, it satisfies this requirement. However, what isn't immediately clear is that KitchenWare only "acts like" a BuildingMaterial when the willAcceptSubstitutes flag of KitchenWareSales is set to false. Following on from this, when making subsequent changes to our architectural fragment we now have to understand that when the orderProduct method of SalesProcess is finding an alternate depot for BuildingMaterialSales (part of the legitimate architecture), it may actually be dealing with sales of KitchenWare for which no substitutes were acceptable to the chef (part of the implicit architecture). Only a detailed understanding of the intricate dependencies between the "hacks" force-fitted on top of our architecture will reveal such subtleties. Subsequent hacks, therefore, require not just an understanding of how they fit into the legitimate architecture, but also a deep appreciation of the impact of, and their impact upon, the layers of implicit architecture hacked in place before them. As each successive *wave* of hacking attempts to circumvent another commonality and variability assumption, it force-fits another form of variability not anticipated in and not accommodated by the system's legitimate architecture.

Successive waves of hacking, then, lead to an increasingly fragile lattice of implicit architectures balanced on top of the system's legitimate architecture, circumventing more and more commonality and variability assumptions at each level. The implicit design underlying such a lattice is, in general, fiendishly difficult to unravel and comprehend. As a consequence, it becomes progressively more and more challenging to construct satisfactory and succinct mental models of the system's architecture with which to evaluate appropriate modifications to the system in response to further requirements change.

In the absence of useful mental models, the "big picture" is abandoned and attention is repeatedly diverted instead to low-level implementation details. When attention is continually focused at the detail level, further waves of hacking ensue, and the system's architecture continually degrades further and further away from anything purposeful. Eventually, the intellectual capacity of developers to understand the increasingly intricate relationships between the expanding successive layers of implicit architecture is exhausted, and at this point *the code is in control*. By now, the architecture is essentially *accidental*, has petrified, and thus substantially resists the reflection of any further requirements change. That is, we now have a legacy system.

4.15. Review

In this chapter we have seen that requirements change is a call for realignment of a solution domain (i.e. an application) with a problem domain (business needs) that has undergone evolution. Where this realignment can be confined to predetermined points of variability, the application's architecture can remain stable. Unanticipated change, on the other hand, is inherently more risky, potentially invalidating architectural commonality and variability. Hacking is an understandable response to the risk inherent in such unanticipated change. Unfortunately, successive waves of hacking lead inevitably to petrified architectures and hence to legacy systems.

Hacking is not purely a consequence of developer whim. Indeed, as we will see in chapter 6, hacking is positively encouraged by many established development methodologies. Starting in chapter 7 we will elaborate emerging alternatives that embrace unanticipated change and encourage perpetually adaptable systems. In particular, chapters 8 through 11 elaborate novel process of Productive Migration, with which we can make a petrified legacy system adaptable again. Chapter 11, for example, demonstrates how the “hacked” architectural fragment depicted diagrammatically in this chapter can be “legitimised” to more directly reflect our revised commonality and variability assumptions. Chapter 12 then elaborates how a newly adaptable system can remain adaptable in the light of further unanticipated requirements change by following a practice of Productive Evolution. First, though, in chapter 5 we shall see that different development methodologies are suited to different levels of developer maturity in terms of how they deal with requirements change. Most importantly, we shall see that an appeal to methodologies aimed at low levels of developer maturity leads to a denial of unanticipated requirements change, to encouragement of hacking practices, and ultimately to petrified architectures and thus to legacy systems.

Chapter 5

Developer Maturity

How beauteous mankind is! O brave new world,

That hath such people in't!

- William Shakespeare, *The Tempest* V, 1

5.1. Preview

In the preceding chapter we saw that petrification results from hacking, wherein unanticipated changes are force fitted into inappropriate architectures. A major realisation from our work is that hacking is often a consequence of following standardised methodological practices aimed at lower levels of developer maturity than are necessary for dealing with legacy systems. In this chapter, we elaborate three levels of developer maturity and observe that there is little methodological support for the highest level of maturity (Adaptation), which is the level at which developers need to work to tackle and prevent architectural petrification[20].

5.2. Maturity Levels

Ongoing observation of, and discussions with, the developers and managers at EDP have led to the realisation that software developers pass through (up to) three levels of professional maturity:

1. **Translation:** At this level the developer is capable of translating models from one notation to another. Primarily, this involves absorbing a reasonably complete implementation specification and realising it in equivalent program code. Alternatively, it may mean rewriting an existing program in a new programming language. At this level of maturity, the developer is steered very closely, with little room for manoeuvre and hence little room for error (other than mistakes in translation, which hopefully inspection and testing will reveal). Developers well versed in translation are perceived at EDP as suited to writing software applications

[20] Note that a recent study [Burd et al. 2001] of the maintenance of several legacy systems supports this point, since it was found that "[when] the best programmers were assigned to maintenance tasks the overall quality of the code tended to improve." This is seen as a "complete reversal of the standard … path of software … maintenance where usually a steady increase in … complexity is identifiable." [ibid.]. Indeed, studies by Swanson and Beath unfortunately indicate that frequently more than 60% of developers assigned to maintenance work are inexperienced new-hires (with more than 25% being students!) [Swanson and Beath, 1989]. Pigoski puts the figure even higher, with his own studies indicating the percentage of "maintainers" who are new-hires to be as high 80% [Pigoski, 1997].

whose specifications are well-known up front, are not expected to change substantially, and have well-bounded interfaces.

2. **Abstraction:** Most developers, having mastered translation, mature to a higher level of competence with abstraction. Here the developer is able to recognise patterns in both the problem specification and the emerging solution implementation, and constructs layered and partitioned models of each to embrace those patterns and thus affect reuse. Particularly evident at this level of maturity is the construction of software architectures to "support" the application. Emerging patterns are taken as points of commonality encapsulated in architectures, where divergence is expressed differentially at predetermined points of architectural variability. We have found at EDP that developers who have achieved maturity in abstraction are trusted to construct software architectures where requirements are not well understood up front but emerge and stabilise over time, with potential requirements change more or less predictable and catered for via architectural variability points.
3. **Adaptation:** At the third level of maturity, developers are expected to deal with considerable uncertainty and to "expect the unexpected". Here, software architectures are considered highly fluid, like Plasticine (as opposed to clay), which remains pliable in the hands of a model maker. Requirements are expected to change both throughout and after initial development and delivery. Architectures are seen only as snapshots taken from a current understanding of requirements, and are frequently completely re-architected, as future changing needs dictate. Furthermore, developers at this level are generally highly experienced and completely comfortable with taking source code developed by others, absorbing its intent, and completely reworking it if necessary. Software is viewed as completely soft, and there is sufficient self-confidence and a sufficiently high level of skill to be able to undertake the (sometimes major) overhauls of an application necessary to reflect totally unexpected directional shifts in requirements. In summary, at this level of maturity the developer is expert at dealing with uncertainty, is able to take the flimsiest initial specification and make something useful from it, and is able to completely redirect that system when the wind blows the requirements in some unanticipated future direction.

5.3. Peaking

Based on our experience at EDP, we see each level of developer maturity as being build upon the preceding level. It appears that being fully versed with translation is an

essential prerequisite to abstraction, which in turn is considered essential to eventually achieving adaptation ability. Passing from one level to the next often takes many years, although in rare cases particularly capable people seem to have passed through all three levels at lightning speed. A rough and ready straw poll at EDP has indicated that only about a half of developers ever reach the highest level of maturity (Adaptation). Perhaps thirty percent of all developers never mature beyond the second level (Abstraction) despite many years of experience, and in a smaller number of exceptional cases (about twenty percent) the developer remains permanently stuck at the first level (Translation). The level at which a developer "peaked" could not be accounted for in terms of longevity of experience, training undertaken, or variety of project exposure. The general belief of those polled at EDP is that personality and, above all, individual aptitude were the main differentiators between those peaking at each of the levels.

Proposing that different developers have different potential in terms of their peak level of maturity is likely to prove controversial. However, more than thirty years of experience appears to have led EDP to the hard-to-refute conclusion that different people do indeed have different aptitudes. Early efforts to "train up" those who "lagged behind" proved unfruitful, and a later effort to "only employ the best" revealed - as many companies must have seen - that there are only so many of "the best" to go around. With experience, EDP has actually learned that all three levels of maturity are valuable to the company. Projects now tend to "find their own level" with the emergence of work suitable to a variety of aptitudes and experience. Interestingly, there appears to be little resentment amongst those peaking at lower levels towards those peaking at the top level of maturity. Indeed, there seems to be a general tacit agreement amongst developers about who is currently at which level: a straw poll at EDP led to remarkably consistent agreement about who peaked where. Perhaps this stems from the fact that EDP has a relatively flat legitimate corporate structure, with "hot shots" more or less emerging "from the ranks" rather than being appointed from on high.

5.4. Dealing With Uncertainty and Instability

We can summarise the different levels of developer maturity as a measure of a developer's ability to cope with uncertainty. At the first level (Translation) a detailed specification leads to a high level of certainty about what is required of the developer. At the second level (Abstraction) an architecture pinpoints areas of potential variability. These are points where requirements are expected to change, where there is some uncertainty about exactly what changes will occur in the future but there is a reasonably

good idea about what those changes might look like (due to some form of domain analysis). At the third level of maturity (Adaptation) neither the current nor future requirements are clear; there is a degree of uncertainty about what is required now, and an even greater level of uncertainty about how stable those requirements will be in the future.

5.5. Supporting Development

There is considerable support both commercially and from academia for dealing with relatively high levels of certainty and stability. For example, at the first level of maturity (Translation), where the specification is both knowable and known, requirements can be captured up-front and expressed with a variety of informal (e.g. English narrative, black-board sketches), moderately formal (e.g. UML [Rumbaugh et al. 1999]]) and highly formal (e.g. Z [Abrial et al. 1980]) notations. Similarly, at the second level of maturity (Abstraction), where requirement are initially unknown but emerge through a domain analysis, there are processes (e.g. RUP [Jacobson et al. 1999]) available with which to elaborate an appropriate stable architecture. Support for these first two levels of maturity is encapsulated in predictive methodologies, described in the next chapter, which work well in environments where there is a low level of uncertainty, as explored in the following chapter.

For the third level of maturity - where uncertainty and potential instability are high – we have found little available to support developers other than their own ability to "think on their feet". This is problematic, since relying on developers' instincts is unsettling to management, yet appealing to inappropriate methodological support aimed at lower levels of maturity leads to denial of uncertainty, which is damaging in unstable environments (such as where business processes change unpredictably). In particular, clinging to a stable architecture over time (as encouraged by Abstraction level support methodologies such as RUP [Jacobson et al. 1999]) can lead to petrified architectures, and hence to legacy systems, by encouraging architectural hacking as outlined in the preceding chapter.

5.6. Relevance to Legacy Systems

How we respond to the level of uncertainty and instability of current, and more importantly future requirements affects our ability to tackle and prevent legacy system petrification. When current and future requirements are clear, we can prevent legacy systems from occurring by appealing to well established software engineering principles (such as forming stable architectures), but when requirements are unclear and unstable

we need to protect ourselves from unwittingly taking measures that ultimately prove to be counterproductive. It is only by adequately supporting developers working at the appropriate maturity level for the degree of environmental uncertainty and instability they face, rather than forcing them to work at the safer levels below, that we can deal effectively with legacy system petrification and prevent it from recurring in the future.

5.7. Review

In this chapter we have seen that developers peak at one of three levels of maturity. It is only at the highest level of maturity that developers are able to cope with the levels of uncertainty prevalent in unstable environments where requirements frequently change in unanticipated directions[21]. In the next chapter, we investigate *predictive methodologies*, which support the lower two levels of developer maturity, and hence anticipate high levels of requirements certainty and stability. In the chapter after that we explore recently emerging work in *agile methodologies*, which offer insight into how we can effectively support development amid uncertainty and instability. Agile methodologies give fresh insights into avoiding petrification. They help keep a system adaptable to unanticipated requirements change, and hence hold great promise for supporting developers tackling and preventing legacy systems.

[21] This does not, however, mean that developers working at lower levels of maturity are redundant. Rather, we have found at EDP that developers working at the Adaptation level are best focused primarily upon "managing" uncertainty for others. Thus, we see the job of Adaptation-level developers as recognising when commonality and variability assumptions are invalid, determining what the new assumptions need to be, and providing adequate and temporarily stable architectural support for them to developers working at the Translation and Abstraction levels of maturity.

Chapter 6

Predictive Methodologies

Our wills and fates do so contrary run
That our devices still are overthrown.
- William Shakespeare, *Hamlet* III, 2

6.1. Preview

In an earlier chapter, we saw that petrification is a consequence of hacking, which is an attempt to stem architectural change in the light of unanticipated requirements change. Hacking can only occur when we have software development beliefs and practices that allow or even encourage it to happen. Ultimately, hacking stems from the belief that software architectures can be pinned down with a thorough prior understanding of commonality and variability points across an essentially stable domain. Prevalent *predictive methodologies*, surveyed in this chapter, promote precisely that belief.

The assumption inherent in predictive methodologies is that all necessary degrees of change can be predicted in advance. This is an assumption subscribed to by developers working at the first two levels of maturity (Translation and Abstraction) outlined in the preceding chapter. Hacking is an unfortunate side effect of an attempt to remain faithful to such predictability despite compelling evidence to the contrary. To prevent currently adaptable systems from degrading into legacy systems we need to abandon the assumptions and practices that lead to hacking, and turn instead to practices (described in later chapters) that welcome and accommodate both anticipated and *unanticipated* change as natural, inevitable, and in fact ultimately manageable.

6.2. Fear of the Unknown

Software is developed using habitual practices, which in their most formal are codified in defined development methodologies. At their least formal, a shared set of tacit behaviours forms an implicit development methodology. A methodology guides the development process, steering the direction it follows through the production of various artefacts.

It has been said that all methodologies are a response to fear [Beck, 2000]. *Predictive methodologies* are a response to the fear that unknown requirements lead to uncertainty with respect to appropriate system architecture. The fear is that as our understanding of

system requirements changes, the architecture necessary to accommodate those changes may also need to change. The underlying belief is that such architectural evolution will get out of hand, leading to chaotic and highly risky development practices. Predictive methodologies strive to control such risk by imposing "a disciplined process upon software development with the aim of making software development more predictable and efficient. They do this by developing a detailed process with a strong emphasis on planning inspired by other engineering disciplines" [Fowler, 2000].

Examining and reflecting upon numerous predictive methodologies (e.g., [DeMarco, 1978; Yourdon and Constantine, 1978; Gane and Sarson, 1979; Eva, 1995; Booch, 1991; Rumbaugh et al. 1991; Jacobson et al. 1999; Graham et al. 1997]) we can see that they share a number of interesting characteristics, as described below:

- **Faith in the process**: they place great faith in a highly defined (albeit somewhat configurable) process. Team members are organised according to specific roles with clearly defined responsibilities on the project, and the process prescribes the activities to be followed by role-players in achievement of their responsibilities.
- **Focus on model building**: they emphasise model building, producing a large collection of specific diagrammatic artefacts codified in a defined notation. Production code is viewed as the final backwards-traceable incarnation of a series of prior consistent models.
- **Focus on model-based communication:** since the process coordinates development in the capture of important project information in detailed models, those models can convey the state of the project across the project team and hence serve as a communication conduit.
- **Anticipate stability:** they expect requirements to be either pinned down up-front or to evolve over time only in previously anticipated ways. Consequently, the emphasis is on the construction of "robust" architectures whose commonality and variability assumptions remain stable over time.
- **Tackle risky change up-front:** they view code changes as expensive and risky and hence strive to tackle risky issues (such as potential requirements change) early on, delaying code production to the later "construction" stages of development and discouraging major code changes once construction is complete.

- **Emphasise management control:** they emphasise a management macro-process that controls and schedules the work of the product team, aimed at preventing the product team from taking a wrong turn.
- **Are repeatable:** they advocate process repeatability. The intent is that by following a repeatable process, the outcome will be repeatable too. Project specific conditions are accommodated via predefined points of process variability, with predetermined "configuration points" for all necessary adaptation.

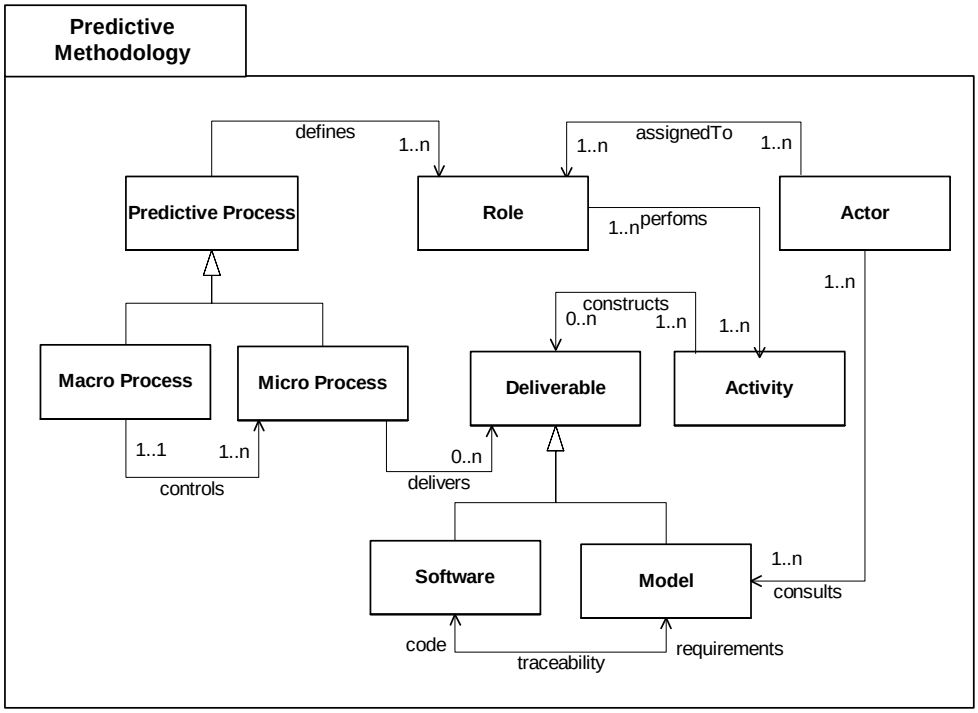

Figure 6.1 – Predictive Methodology

As depicted in Figure 6.1[22], predicitve methodologies promote a management macro process controlling closely the underlying micro processes that deliver product. Both the managment macro process and the managed micro processes prescribe clear-cut roles for abstract actors, specifying the precise activities they need to perform in the construction of deliverables. Those deliverables are ultimately software code and speculatively communicative requirements models, with direct traceability between the two.

[22] Figure 6.1 captures only essential features of predictive methodologies – a complete predictive methodology would naturally require far more elaboration. The aim here, though, is not to be exhaustive but rather to provide a sufficient comparative basis with which to evaluate alternate forms of predictive methodology (see Figures 6.2 and 6.3 below), and contrast them (in the next chapter) with agile methodologies, which do not attempt to be predictive.

6.3. Application-Oriented Predictive Methodologies

We have seen that predictive methodologies stem from a terrifying fear (human petrification) that poorly understood requirements lead to unpredictable consequences in later development. Since unpredictability is inherently risk laden, the earliest predictive methodologies, which we term *application-oriented* predictive methodologies, (such as [DeMarco, 1978], [Yourdon and Constantine, 1978], [Gane and Sarson, 1979], culminating perhaps in SSADM [Eva, 1995]) aimed to introduce predictability by completely pinning system requirements down up-front, before development of other artefacts (such as designs and program code) began. The intent here was to construct an essentially immutable set of requirements for an effectively immutable application (see Figure 6.2). By pinning requirements down up-front, we would have a better chance of delivering a system that satisfied those requirements, and hence reduce the costly maintenance burden inherent in last minute requirements changes[23]. We could also prevent the need for frustratingly expensive bug fixes stemming from early requirements misunderstanding[24].

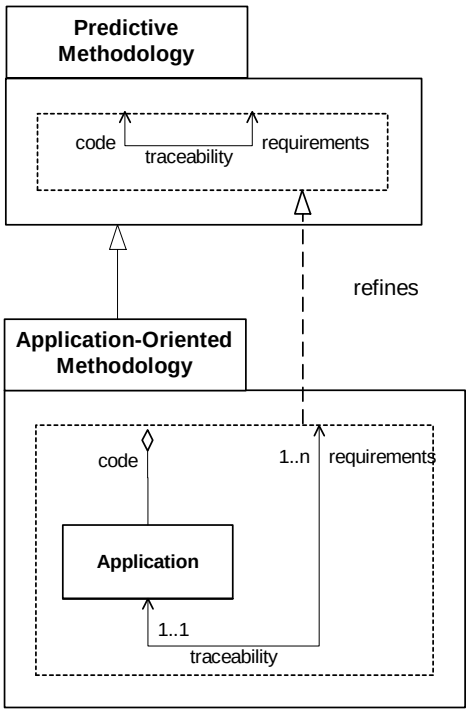

Figure 6.2 – Application-Oriented Predictive Methodology

[23] That is, we can reduce the maintenance burden by delivering systems that do not require maintenance! More specifically, many application-oriented predictive methodologists claimed that down-stream changes to requirements were phenomenally expensive compared to, say, code changes (perhaps one hundred times more expensive). Consequently, by "disallowing" post-development requirements changes we would, it was claimed, save ourselves considerable financial expense!

[24] Unfortunately, however, extensive studies show that only 20% of total maintenance spending actually relates to bug fixing [Lientz and Swanson, 1980]. The remaining 80% is inevitably consumed in response to ongoing technological and business requirements change [ibid.].

In Figure 6.2, above, we see that application-oriented methodologies refine the software deliverable of predictive methdologies (c.f. Figure 6.1) to be a single application, directly traceable to its specific requirements.

Application-oriented predictive methodologies seem well suited to developers working at the first level of developer maturity (Translation), where requirements change is not anticipated. Unfortunately, when such methodologies are employed, any requirements changes that do occur, either during or after initial development, become problematic.

6.4. Architecture-Oriented Predictive Methodologies

Application-oriented predictive methodologies suffer from the belief that requirements can be pinned down definitively for a given application. This is true only for a limited range of applications. Missing here is explicit support for any degree of flexibility necessary to accommodate future requirements change. To permit such flexibility, the ambition of contemporary predictive methodologies (predominantly object-oriented methodologies, such as Booch [Booch, 1991], OMT [Rumbaugh et al. 1991], RUP [Jacobson et al. 1999], and OPEN [Graham et al. 1997]) is rarely to identify completely static application requirements, but rather to pursue a thorough domain analysis from which all points of potential variability (change) are well understood. From this deep understanding we can, it is believed, construct essentially stable software architectures with all necessary points of variability built in, with the impact of possible future requirements change being predictable and hence manageable.

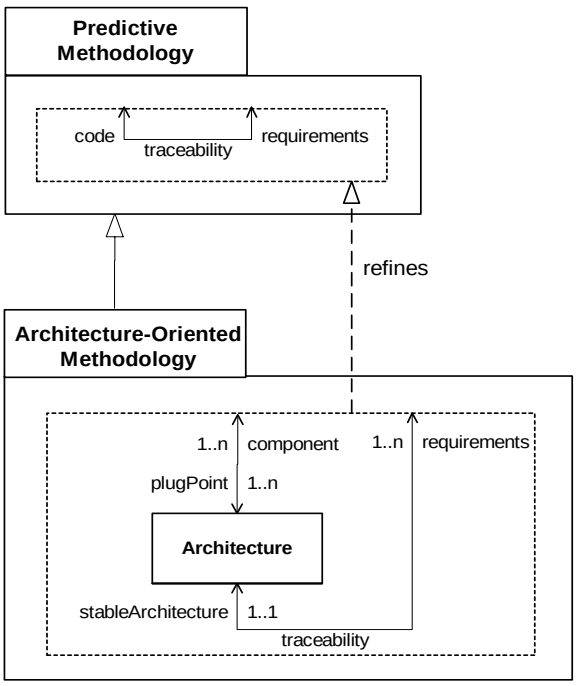

Figure 6.3 – Architecture-Oriented Predictive Methodology

Such *architecture-oriented* predictive methodologies, then, pin down architectures, not applications (see Figure 6.3, c.f. Figure 6.1). They aim to restrict future requirements change to that requiring only localised changes (i.e. maintenance) in the application at predetermined variability points, and (it is believed) eliminate the need for risky architectural change (evolution) at previously determined points of commonality. Architecture-oriented methodologies seem well suited to developers working at the second level of maturity (Abstraction) wherein stable abstractions emerge from requirements which may then be mapped to stable architectures accommodating all necessary degrees of freedom for future (anticipated) requirements change.

6.5. Architectural Pliability

Unfortunately, architecture-oriented predictive methodologies see an architecture's points of commonality and variability not as a journey but as a destination. Such methodologies advocate architectures that are *variable* but not *pliable*. They are variable in that they can be adapted (maintained) at the predetermined variability points revealed during domain analysis, which are seen as accommodating all necessary degrees of future requirements change. Pliability, which is missing here, would allow architectural evolution of current commonality points (i.e. at places previously assumed to be immutable) into new points of variability (and, indeed, vice versa). Pliability, then, enables us to continually mirror shifts in domain commonality and variability resulting from ongoing requirements change

6.6. Unpredictable Change

A major problem with predictive methodologies is that (as seen in earlier chapters) in many contexts much future change is inherently unpredictable, and hence ambitions for a definitive and stable set of commonality and variability points are ill founded and doomed to fail in the long term. In other words, as we observed in chapter 4, a domain analysis can result in only a snapshot of what has been learned from past experience about commonality and variability across a domain. This is fine for domains that are essentially stable, but for those that undergo continual and often unpredictable change it is wholly inadequate. Unanticipated changes to a domain (possibly from unforeseen external forces) have the great potential to upset and invalidate the commonality and variability set revealed during a domain analysis. Thus, unanticipated changes have the potential to destabilise a system's architecture cast from such an assumption set.

6.7. Architectural Evolution

This misguided belief in the universal validity of essentially stable architectures increases the likelihood of late delivery (since unexpected requirements change during development is surprising and hard to respond to) and of effectively immediate petrification (since future unexpected requirements change will be surprising and hard to respond to). When we allow architectures to become misaligned with the changing commonality and variability points of an evolving problem domain, petrification will prevail. To prevent petrification - and hence to address legacy systems effectively - we need to let go of our fear of unexpected requirements change. Only when we accept that much requirements change is unpredictable can we accept that architectures must co-evolve with that change if they are to remain valuable to us. Embracing architectural evolution, then, is a central pivot of an effective strategy for preventing legacy system petrification. This requires developers to work at the third level of maturity (Adaptation), wherein we abandon naïve appeal to complete predictability and instead learn to live with and work with potentially high levels of uncertainty and change.

6.8. Review

Faith is belief without evidence. Predictive methodologies compel us to believe faithfully that we can create system architectures that remain essentially stable over time. Complaints from developers about clients "moving the goalposts", as if that was a sin, are symptomatic of this belief, as are frequent excuses that a project is late because "the requirements keep changing under us". Viewing unexpected requirements change as heresy is denial of the undeniable evidence that certain types of change are essentially unpredictable. A recent example of a major widely unpredicted requirements change is the explosive demand for e-commerce applications over the Internet. Such major unanticipated requirements change can be extremely destabilising to a system's established architecture. Even small-scale unanticipated requirements change can have the same effect. It is commonplace for customers to hear "that might sound like a small change, but the system wasn't designed to handle things like that and it’s going to be months of work for us."

In summary, unanticipated changes can completely undermine a system's architectural commonality and variability assumption set, and hence introduce significant risk to a project. Predictive methodologies attempt to mitigate this risk by behaving as if unanticipated change doesn't exist. The belief that we can achieve predictability of both requirements and their potential for change is, of course, flawed and dangerous. It

inevitably leads to system petrification as developers struggle to hack square pegs into round holes. To avoid petrification we need to let go of the compelling urge to achieve predictability and accept that we work in an inherently unpredictable world. Rather than imposing futile ambitions for predictability, we must turn instead to beliefs and practices that welcome and accommodate both anticipated and unanticipated change as natural, inevitable, and in fact ultimately manageable. This is the focus of the next chapter.

Chapter 7

Agile Methodologies

Our remedies oft in ourselves do lie

Which we ascribe to heaven.

- William Shakespeare, *All's Well That Ends Well* I, 1

7.1. Preview

In the previous chapter we saw that predictive methodologies are frequently a prescription for petrification. A major problem with predictive methodologies is that they manage the risks of anticipated change but in doing so they magnify the risks of unanticipated change. When unanticipated changes do occur, immutable commonality and variability assumptions are immediately undermined leaving the developer with little choice other than to hack away at the system's supposedly stable architecture. As we saw in chapter 4, as a result of hacking the developer imposes layer upon layer of convoluted implicit architecture on top of the system's legitimate architecture in a heroic effort to mutate a supposedly immutable architectural base. The inevitable result is system petrification.

In this chapter we elaborate an approach to development wherein there is an undercurrent of continual feedback, perpetual reflection, and ongoing adaptation of the methodology itself in response to emerging and unanticipated requirements change. The aim is to *embrace change*[25], work with change rather than against it, and ultimately use change to our advantage.

This chapter is a relatively lengthy one, but necessarily so since it lays out principles and practices to which later chapters appeal in the Productive Migration of legacy systems to adaptable systems, and in the continued Productive Evolution of those adaptable systems in the light of further requirements change.

7.2. Silver Bullets

Reliance on predictive methodologies may possibly stem from a belief that the only alternative to them is *Wild West programming*, wherein cowboy team members are left completely unguided, pursuing their own chaotic whims, resulting in unmanaged and unmanageable catastrophe. Of course, neither the predictive nor the Wild West

[25] To borrow a term from [Beck, 2000].

approach is acceptable if we are to deal effectively with the difficult task of tackling both anticipated and unanticipated requirements change. "An ad hoc approach gives the analysts and programmers an excuse not to think, while the bureaucratic approach give managers an excuse not to think." [Lynne Nix, quoted in [Highsmith, 1999b]]

It transpires, however, that there is another way. Although it is true that "there are no silver bullets" [Brooks, 1987], there are sometimes “heroic” [Bach, 1995] Lone Rangers [Weinberg, 1994] who can rescue us from catastrophe, and are armed with "arsenals of bullets for different situations" [Highsmith, 1999b]. Lone Rangers recognise that we need to remain flexible to the diverse needs of diverse contexts, and arm ourselves with and continually cultivate diverse strategies for tackling them effectively. Agile methodologies – which we survey in this chapter - strive to nurture individual Lone Rangers into jelled teams, collaborating to remain responsive to evolving contextual needs. The realisation is that although an adaptable architecture cannot rescue us from unanticipated requirements change, a jelled team guided by an agile methodology can.

7.3. Expect the Unexpected

In her excellent book Plans and Situated Actions, Lucy Suchman presents compelling evidence that up-front prediction of complete, accurate, and essentially unchanging plans is rarely achievable [Suchman, 1987]. Rather, many of the necessary actions, and changes within them, materialise only when we have already begun pursuing the preliminary course we have to hand. That is, we can only begin to get a grasp on some of the questions when we have already attempted to struggle with some of the answers.

"[P]lanned purposeful actions are inevitably *situated actions* … [i.e.] taken in the context of particular, concrete circumstances … [T]he circumstances of our actions are never fully anticipated and are continuously changing around us. As a consequence, our actions, while systematic, are never planned in the strong sense that cognitive science would have it. Rather, plans are best viewed as a weak resource for what is primarily *ad hoc* activity." [Suchman, 1987]

Except for in very well understood problem domains, then, effective plans are not the result of up front planning (other than in very sketchy terms), but rather plans evolve naturally in response to changing conditions along the way. This does not mean, however, that we have to "wing it" as we go along. Rather, we need to learn from projects that have already succeeded in navigating an unfolding path, embrace heuristic beliefs that have already proven effective, and capture and cultivate new effective heuristics as they emerge.

In this chapter we shall see that we can indeed both have our cake and eat it; we can control risk yet also embrace unexpected change. This, however, requires discipline; not a discipline imposed by a dictatorial, bureaucratic, predictive methodology, but self-discipline supported by a lightweight methodology guiding and encouraging creative effort amid constantly changing needs.

7.4. Agile Methodologies

Numerous methodologists have gathered empirical evidence "in the trenches" showing that successful projects tend to follow practices very different from those widely advocated by popular predictive methodologies (see, for example, [Bach, 1999c] for one personal high-level overview of differences between theory and practice). Indeed, Alistair Cockburn - a highly regarded methodologist - spent several years interviewing wide ranging project teams, each of which had utilised a defined predictive methodology, and has concluded that not a single project was successful [Cockburn, 2000c]. Particularly surprising here are his minimal requirements for a project to be deemed successful: Firstly, some product must eventually have been shipped. Secondly, the project team must be willing to utilise the same methodology again.

Disappointment with the success of predictive methodologies has resulted in the emergence of several "new" methodologies, based upon empirical evidence of what actually has worked in practice. These "new" approaches (Adaptive Software Development [Highsmith, 1999b], Crystal [Cockburn, 2001b], Scrum [Beedle et al. 2000], and Extreme Programming [Beck, 2000] being the most prominent) are generally termed *agile methodologies*[26]. Agile methodologies strive to provide just enough rigour to support progress, whilst remaining sufficiently lightweight and adaptive to respond to continually changing circumsances. Our own experience at EDP has revealed that it is precisely this type of approach that is appealed to by developers working at the highest level of maturity (Adaptation). By embracing the principles of agile methodologies developers can work productively in the dark without falling flat on their face.

7.5. Core Principles

Analysis of prominent agile methodologies (based primarily upon their published work) reveals common underlying principles that differentiate them from predictive methodologies. Predictive methodologies put faith in highly defined repeatable

[26] Indeed, a number of prominent “agile methodologists” have recently united to form the Agile Alliance (see www.agilealliance.org), aiming (perhaps) to unify their approaches, or (at the very least) to agree upon common principles.

processes, with people as substitutable role-fillers. Agile methodologies, on the other hand, put faith in the people themselves. The process is there to guide people, not to control them, and it is left to the skill and judgement of the people themselves to determine when the process is and is not working, and when it needs adapting to support the team more effectively. Expanding on this, we can say that agile methodologies:

- **Expect continual change:** rather than expect requirements to be pinned down up front, they expect requirements to evolve over time, and keep the customer involved along the way to assist with their continual elaboration.
- **Accept that software is soft:** they are not afraid to rewrite things that were previously thought to be "finished" when changing requirements call for possibly radical rewrites.
- **Are low on ceremony**: unlike heavyweight predictive methodologies, agile methodologies tend to be very lightweight, with few rituals to be followed slavishly. Instead a small number of heuristic practices emphasise guided self-discipline.
- **Are low on bureaucracy**: they tend to de-emphasise document production and focus instead on working code as the primary work product, viewing all other artefacts as only valuable if they contribute to the delivery of working code.
- **Catch mistakes early:** rather than leave testing and delivery to the very end, there is a strong emphasis on continual testing, demonstration, and deployment to give early feedback and allow immediate correction where necessary.
- **Encourage empowerment:** management and clients are expected to give strong and continual input in terms of requirements, and satisfaction with deliverables, but to give significant leeway to the development team's ability to meet those requirements in the way that best suits them.
- **Believe in emergence:** rather than expect major up front planning, prediction, and design, there is recognition that a jelled team achieves effective emergent discovery and invention throughout the whole course of development by building upon what they do know to learn about what they don't know.
- **Focus on face to face communication:** rather than assuming that “the process” will coordinate development, people are expected to coordinate and steer their own work and share knowledge and concerns by continually discussing and evaluating progress with one another.

- **Are reflective:** unlike fully defined repeatable methodologies, agile methodologies encourage continual reflection upon their effectiveness and expect continual adaptation of the methodology itself as the needs of the project team evolve.

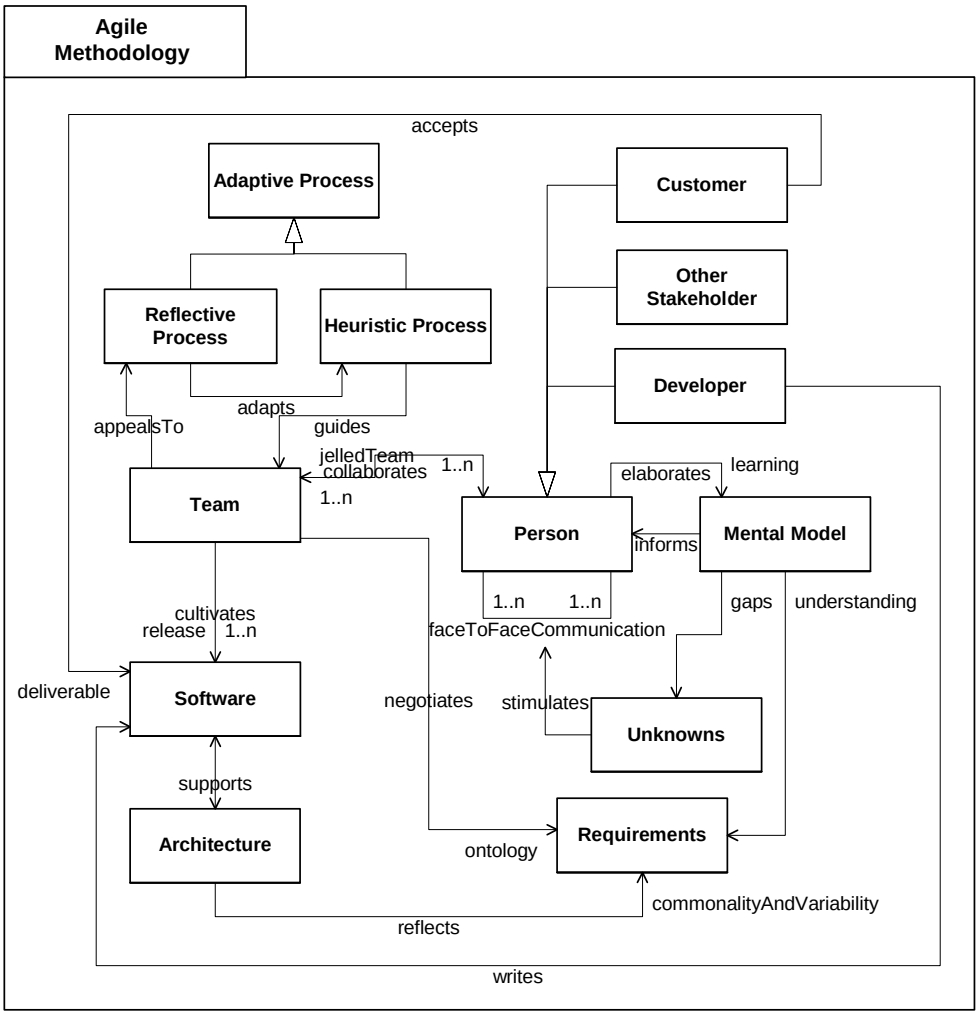

Figure 7.1 – Agile Methodology

Figure 7.1 depicts the main common principles of agile methodologies diagramatically. Thus, we see that the focus is upon reflective jelled teams continually communicating to share evolving mental models of emerging requirements, and adapting heuristic processes to cultivate a progression of software releases (underpinned by evolving supporting architecture) acceptable to customers.

Agile methodologies, then, abandon the idea that requirements can be fully understood or their nature predicted up front. Instead, requirements are allowed to emerge, and indeed change, over the course of development. In general, a project is addressed a small chunk at a time, with requirements fleshed as far as possible (but no further) for the current chunk, which is then developed in a short burst of highly collaborative work. At the end of each short chunk, a demonstration is usually given to the customer,

progress thus far is reviewed, adjustments are made, and the team refocuses on re-negotiating requirements for the next chunk of work in the overall project.

There is an undercurrent of continual reflection not just upon progress of the project, but also upon the effectiveness of the methodology being followed. The team continually evaluates feedback from the project, striving to tailor the methodology to the emerging needs of the team, the project chunk with which they are currently involved, and the general direction in which the project as a whole seems to be heading. Without such continual adaptation of the methodology, there is a risk that the methodology itself would become petrified in the light of unanticipated emergent project needs.

Despite the commonalities across agile methodologies, there still remain important differences between them. For example, some agile methodologies (e.g. ASD and Crystal) focus heavily on the philosophical and sociological aspects of effective teamwork, whereas others (e.g. Scrum and XP) are less philosophical and more pragmatic, promoting specific practices that have proven effective. It has been our experience at EDP (see later chapters) that no single published agile methodology is sufficient for dealing with our particular concern, which is legacy systems. Rather, we have over the past two years tried out the individual agile methodologies and picked from them what has worked for us and eliminated what has not. This blend of “best practices” has evolved throughout the course of our work, adding where necessary additional practices to fill shortcomings in the existing approaches as we came across them.

Before we explore this “tailored blend” in detail (i.e. Productive Migration, chapters 8 through 11, and its follow on process of Productive Evolution, chapter 12), we need to focus more closely on the specific variations among the prominent agile methodologies. This detailed review (below) is intended to illuminate specific principles and practices to which we later appeal in the refinement of agile methodologies specifically to address and prevent legacy system petrification.

7.6. Adaptive Software Development

Jim Highsmith's Adaptive Software Development (ASD) approach [Highsmith, 1999b] argues that traditional rigid processes stifle the creativity necessary to respond effectively to unpredictable change. ASD is underpinned by the belief that such creativity can thrive only "on the edge of chaos", with a lightweight process there to provide just enough rigour to maintain balance on that edge [Highsmith, 1999c; Highsmith, 1998].

Living on the edge of chaos requires a shift away from command-control management and a move towards leadership-collaboration [Highsmith, 1999a]. The role of management shifts away from dictating daily tasks and instead focuses upon leading (but not imposing) the cultivation of an environment that encourages effective collaboration and ongoing process adaptation. This, argues Highsmith, leads to teams burning with inner fire. [Highsmith, 1999b]

Highsmith sets low goalposts for his definition of project success: "First, the project gets shipped. Second, the product approaches its mission profile ... And third, the project team is healthy at the end." [Highsmith, 1999b]. This final point is crucial; unlike predictive methodologies, agile methodologies do not view people as interchangeable cogs slotted into a machine, but as living organisms that need to be nurtured and cared for.

In Highsmith's own (extensive) project experience, fostering a healthier team is best achieved by unleashing emergent order (as opposed to imposed order) [Highsmith, 1998; Highsmith, 1999b]. Emergent order grows out of a project team that collaborates in pursuit of a common goal, adapting their collective behaviour rapidly in response to, and learning from, the evolving needs of the project.

ASD empowers the project team to “roll-their-own” development process as the particular (and often unexpected) challenges of the project unfold. In particular, ASD encourages a "Speculate-Collaborate-Learn" development lifecycle. Firstly, since future outcomes are frequently unpredictable, large-scale up-front planning is abandoned as ineffective, and instead developers speculate on what is an appropriate next step mission to undertake based upon their current understanding. Having committed to a shared mission, developers collaborate towards achievement of that mission and delivery of its intended work products, sharing their creation and discovery as they go. Finally, they (and, indeed, customers and other stakeholders [Highsmith, 1997]) learn from that collaboration about where they were successful and where they were not (via team reflection and discussion, customer focus-group feedback, software inspections, post-mortems, and other techniques), shifting accordingly their focus for the next mission speculation.

Note that at all points throughout the Speculate-Collaborate-Learn lifecycle, there is acknowledgement for the occasional need to "break out" of the loop, to explore the

unknown before it is appropriate to come back again to the comfort and order of a regular lifecycle[27].

7.7. Crystal

Alistair Cockburn has discovered after several years of project debriefings that the more heavyweight a methodology is, the more difficult it becomes for developers to adhere to it. Consequently, he advocates the lightest possible methodology that a project can get away with. His "ultralight" family of Crystal methodologies advocates reducing bureaucracy, increasing self-reliance between collaborating team members, getting code shipped out of the door quickly and frequently, and giving the team continual user feedback on the code they are delivering. By focusing upon delivering code, and being unencumbered by other deliverables, developers can move quickly, and change direction quickly, which means that they can deliver satisfactory code more quickly.

Crystal recognises that teams working closely together, and communicating well, cultivate an emergent and shared understanding of the problem and solution spaces, and a shared vocabulary with which to discuss them. It is this communication grounded in an emerging shared experience - rather than any mandated bureaucracy - that keeps the project on track and fast paced: "People who know their architecture and design patterns will have very nice, short ways of describing their system to each other. Drawing lots of classes and relationships simply cannot replace this communication." [Cockburn, 2001a]

Crystal embraces a metaphor of software development as a cooperative game, wherein participants help each other reach the end of the game, which is the delivery of software. The whole emphasis is upon a team working closely together and doing whatever it takes to get the required software to the customer. All work products other than shippable code (requirements documents, designs, test plans, etc) are only of value if they help the team move forward as rapidly as possible. Any other intermediate work products (which Cockburn terms *markers*) are there to remind members of the team of the current state of play in the cooperative game, to inform other team members of the same thought, and to incite or inspire them in getting to the next move in the game. The production of markers that do not contribute to team effectiveness is considered detrimental bureaucracy, best avoided. At the end of the current game, the delivered

[27] In support of this, a thought-provoking article [Mitleton-Kelly and Papaefthimiou, 2001], from another project in the same SEBPC programme as the work underpinning this thesis, convincingly explains the complex theory of why emergent order can only occur under conditions of self-organisation, free from management controls and other pre-suppositional constraints.

product and the associated residual markers "inform and assist the players of the next game [which] is the alteration or replacement of the system, or creation of a neighboring system." [Cockburn, 2001a]

Markers are considered central to Crystal. In particular, there is a strong emphasis on the belief that there should be no legislation with respect to the form that markers take. The value of markers lies only in their effectiveness at increasing the effectiveness of the project team [Cockburn, 2001b]. "The unfolding dance of requirements and design relies on the team and users putting down evocative and memorable markers of their current understandings, so they can move forward and refer back. These markers need not have great permanence or formality, since they serve to forward the conversation … One of the forms of permanence is inside people's heads. Paper is a tedious and actually pretty poor way to communicate a design. Far better is to talk to another person in front of a whiteboard and put it directly into their head, checking as you go." [Cockburn, 2001a]

Markers are seen in Crystal as "props" which help "to draw the people out of themselves to share and generate new ideas. The props need only be sufficient to their tasks. Paper cut-outs, sticky tape, mock-ups, sketches on whiteboards, mini-plays -- all may work as props." [Cockburn, 2001a]. Each marker need only be complete, or rigorous, enough to fulfil it purpose - which is ultimately to help the project team ship satisfactory code to the user.

Markers capture a team's evolving shared understanding with respect to the customer's emerging needs and how to satisfy them. Teams communicate to evolve this shared understanding and thus to increase the effectiveness of the team as a whole. The form that such communication takes, and the nature of the markers used to record its results, will depend heavily upon the prior experience of the team, the culture in which they work, and the nature of the problem and solution domains with which they are working. Consequently, Cockburn asserts, it is not possible to dictate a single defined methodology, appropriate for all projects [Cockburn, 1999a]. Rather, the team should be given sufficient freedom to work out for themselves the forms of communication, markers, and internal structure that work most effectively for them [Cockburn, 1999b]. In particular, the team should be permitted to evolve their own methodology as the project unfolds, adapting and reinventing it as their shared understanding increases and their intra-team communication mechanisms become more efficient along the way [Cockburn, 2000a].

7.8. Scrum

Scrum [Schwaber, 1995; Beedle et al. 2000] is the result of empirical observation [ControlChaos, 2000b] of high-productivity software development organisations, capturing their best practices [ControlChaos, 2000a]. Analysis has revealed that productive software development groups follow empirical processes guided by heuristic practices (rather than external mandates) responsive to unpredictable and continual environmental feedback along the way [Schwaber, 1996]. Thus, Scrum views development as an unfolding experiment [Schwaber, 1996] immersed in and focused upon responding to such an unpredictable and chaotic environment [ControlChaos, 2000a].

Scrum consists of three distinct phases. The first phase (termed Planning and System Architecture) prioritises both known and uncovered risks and backlogged requirements, assigns resources, envisions an overall architecture, identifies a target operating environment, and sets a final completion date for the product. This phase is kept deliberately brief, in recognition that much may change throughout the course of development [ControlChaos, 2000c].

The second phase (Development) tackles work in a succession of small increments called Sprints, each staffed by a small team with a focused goal and lasting no more than a few weeks. A Sprint team is led by a Sprint Master, who keeps team members' productivity high by looking out for risks and removing emerging productivity impediments. Scrum requires an atmosphere of trust wherein management is willing to hand over daily control of the project to the Sprint teams. Such "empowerment" eliminates the impediments to progress inherent in continual meetings with management, generation of reports justifying decisions, and awaiting authorisation to proceed, and thus frees the team to spend time being productive [ScrumOnline, 2000b]. The role of management, then, "flips" [ControlChaos, 2000d] from micro-controlling the team to supporting them. Thus management focuses on general strategy (product envisioning - where are we going) rather than tactics (product realisation - how will we get there) and eliminating emergent impediments that hinders team productivity [ScrumOnline, 2000a].

A Sprint team is free to adopt or cultivate whatever process helps it realise its goal most effectively. Without deep immersion within the needs of an individual Sprint, any methodological recommendation can only be speculative.

Throughout Scrum, a backlog of pending Sprints is continually cultivated and prioritised via an ongoing process of negotiation and prioritisation, as new requirements emerge and old ones either fade away or mutate. Without a negotiated backlog list, there is a tendency for every feature request to immediately jump to the top of the priority queue, leading to constant, direction-less, and unproductive interrupt-driven context switching denying developers sufficient time and focus to sink their teeth into any one problem [ScrumOnline, 2000a].

Sprints are driven by short Scrum meetings, held daily, wherein the team articulates what progress has been achieved since the last Scrum, what progress has been blocked, and what needs to be done next. Scrum meetings emphasise team learning [ControlChaos, 2000e].

Daily Scrum meetings have been described as "the thermometer into a chaotic situation, providing a daily view into the dynamics, morale, progress, and overall situation of a work effort" [ScrumOnline, 2000a]. Scrum meetings encourage development team values. In particular, they [Beedle et al. 2000]:

- Promote a sense of urgency, so that nobody falls into the complacency of long deadlines.
- Promote sharing of knowledge and experience.
- Encourage dense communication.
- Allow "warts and all" honest communication, since bad news is not punished but rather looked upon as an opportunity to learn and react. This contrasts with traditional periodic reporting, which is often sanitised to deliver only good news to critical managers, leaving undisclosed nasty surprises bubbling beneath the surface.

At the end of each Sprint, customers and managers are given demonstrations of the Sprint's results (almost always a piece of working code) to give the team much needed feedback to be responded to in subsequent Sprints. This continual progression of Sprints underlies what has been termed the "drum beat of regular incremental deliverables" [ScrumOnline, 2000a].

When a succession of Sprints has sufficiently reduced project risk and realised enough features from the backlog for the product to be released, we enter the final (Closure) phase. Closure involves completing Regression Tests, developing training, support, and marketing material and identifying appropriate changes to be backlogged for Sprints in the next release.

7.9. Extreme Programming

Extreme Programming (XP) is about balancing the rights of customers and developers to ensure that the best possible value product is being delivered at any given time. Underpinning XP is acknowledgement that only so much work can be achieved in a given time. Demanding more than is possible will only lead to disappointment. XP involves the customer focusing upon continually refining a wish list of required features (termed *stories* in XP), and prioritising them in terms of *value to the business*[28]. Even bug-fixes are prioritised and scheduled with stories "so the customer can choose between fixing bugs and adding further functionality." [Beck and Fowler, 2001]

Making business-value prioritisation the iteration driver (rather than, say, risk) keeps the customer in control of what is delivered to them. This also prevents the inclusion of developer-inspired “technical” stories which cannot be tied back to business value; "customers can't prioritise what they don't understand" [Jeffries et al. 2001]. This approach also ensures that only stories giving best business value are delivered first. Late schedule pressures can then only impact stories that deliver lower business value.

Stories are short outlines of required features, forming "promises for conversation" [Jeffries et al. 2001] between the customer and developers. Purposefully, they are not intended to be complete specifications, since the customer will learn as much as developers as the project unfolds.

The highest priority stories are grouped into the scope of the next *release*. All other stories are deferred for later releases. Each release lasts (ideally) about two months after which the delivered functionality (business-value) is deployed "in the field". To enable mid-release corrections, releases are broken down into a series of one-to-three-week *iterations*. Short daily "stand-up" meetings ensure that all participants in an iteration have a chance to share concerns, offer help, and so on [Beck and Fowler, 2001].

Programmers continually (re-)estimate how long it will take to implement each story by giving them relative difficulty ratings (in units we term *clicks* at EDP), then estimating the calendar time for completion of each story according to the time required for stories of similar difficulty in the past. Throughout development they learn more about (and hopefully improve upon) their "velocity" (i.e. productivity) with respect to time required

[28] This is a particularly compelling yet surprisingly novel strategy since “spending on IT is the largest single element of capital investment for most enterprises, [but] few of the executives approving these multimillion dollar investments have a clear idea of the results that they expect to get, or whether they actually achieved the benefits when the money is spent.” [Thorp, 1998]. Indeed, there is mounting evidence of an “Information Paradox” wherein there has been, thus far, little correlation between IT spending and demonstrable business benefit [ibid.].

to implement stories according to difficulty level. This increases the accuracy of revised and future delivery estimates for stories of similar difficulty.

For each story, the customer writes automated Acceptance Tests that when satisfied indicate story completion. Acceptance tests effectively encapsulate specifications for stories and thus constitute what XP advocates see as "the ultimate in traceability" [Beck and Fowler, 2001] from code back to requirements.

When implementing stories, developers always work in pairs. "Two programmers working together generate more code, and better code, than the same two programmers working separately. They can work longer without getting tired, and when they're finished, two people understand everything, instead of understanding just half of everything" [Jeffries et al. 2001].

Developers break stories down into short programming tasks, termed development *episodes* [Beck, 2000], each lasting up to three days. During an episode, a pair first writes a series of unit tests that are bound to fail, then writes code that makes the tests succeed. The pair then simplifies the code (via *refactoring* [Fowler, 1999b]) to improve its "internal quality" [Beck and Fowler, 2001], then reruns the tests to ensure they have done no harm. Test-satisfying code is immediately re-integrated into the team's shared code repository to ensure that everybody is working with the latest code and to minimise editing conflicts. Furthermore, the whole team collectively "owns" all of the code and is free to adapt it so that nobody is "forced to work around a deficiency in one object by hacking up another" [Jeffries et al. 2001].

Every day or two a nominated "tracker" passes from pair to pair asking how they are doing compared to their estimates. Progress in XP is not seen as something expressible in "status reports" emailed to management. Rather face-to-face communication and "the personal touch" are seen as "essential to ... monitoring progress" [Beck and Fowler, 2001]. Emergent delays that cannot be managed by the pair involved are raised to team level for possible resolution. Failing that, the customer is brought in to re-negotiate scope and consider dropping some stories (which will necessarily be lower priority than those already released).

One controversial aspect of XP, encapsulated in the slogan "You Aren't Going to Need It" (YAGNI), is to only program for the current story's requirements rather than architect-in up-front flexibility for the future. After a little reflection, though, it is clear that XP, by appealing to ongoing refactoring to continually adjust the system's architecture, ensures that it has precisely the flexibility it currently needs. This is in

contrast to predictive methodologies, where up-front domain analysis *speculates* required flexibility for a given commonality and variability assumption set. That assumption set may ultimately prove wrong as requirements unfold. XP, on the other hand, is effectively performing a *perpetual* domain analysis, continually evolving the architecture to encapsulate precisely the required points of commonality and variability at any given time. In essence, architectural *speculation* has been replaced with ongoing architectural *certainty*. Thus, XP holds great promise for preventing petrification by ensuring that the currently ideal architecture unfolds along with and in response to the changing requirements it needs to accommodate[29].

7.10. Review

Since the nature of an unanticipated requirements change is, well, unanticipated, we cannot predict up front how we will handle it. Consequently, we need to appeal to a methodology that is responsive throughout the whole of development to emerging requirements change. Agile methodologies "expect the unexpected" and respond accordingly. They accommodate both anticipated and unanticipated requirements change by allowing the path being taken during the course of development to be continually adjusted according to continual feedback along the way. Agile methodologies have the potential to prevent petrification by nipping it in the bud, and continually steering architecture in the direction in which requirements are moving.

An important realisation elaborated throughout the rest of this thesis is that an agile approach is not only helpful in preventing petrification; it can also help us to deal with a system that is already petrified. The next few chapters, then, reveal a novel agile methodology, termed Productive Migration, via which a petrified system can be migrated to an adaptable system. Productive Migration borrows heavily from the agile methodologies surveyed here, but tailors them specifically the needs of petrified legacy systems (something we have not seen done before). A later chapter adds Productive Evolution, whereby an adaptable system can be kept responsive to continual requirements change, ensuring that it will never again degrade into a legacy system. Productive Evolution appears close in spirit to the agile methodologies surveyed here (since its aim is to maintain system adaptability, rather than tackle systems that are

[29] Interestingly, the XP community do not refer to supportive and parallel work by Griswold, whose Just-in-time-Architecture [Griswold, 1996] is strikingly similar to YAGNI, and is motivated by the realisation that up-front architectures are approximately three times more expensive to develop than emerging architectures (quoting a study by DeMarco), and also by Parnas' reservations about his own (influential) up-front program family design approach as being too expensive unless we are certain about current and future family needs [Parnas, 1976].

already petrified). Nevertheless, Productive Evolution in detail is more closely akin to Productive Migration than any prior agile methodology since it aims to carry on the specific good practices that Productive Migration has already established.

Chapter 8

Productive Migration

I have seen a medicine

That's able to breathe life into a stone.

- William Shakespeare, *All's Well That Ends Well* II, 1

8.1. Preview

This chapter is essentially an overview chapter, "setting the scene" for the chapters that follow. Here, we describe how we can embrace many of the lessons from the agile methodologies described in the previous chapter to tackle legacy system petrification. Note however, that published agile methodologies are geared towards the development and maintenance of new information systems. As far as we have been able to tell, the advocates of agile methodologies have not addressed their deployment within the context of information systems that already exist, and more specifically within the context of legacy systems[30]. This thesis aims to fill that gap.

Over the past two years dealing with legacy systems at EDP we have "tried out" the published agile methodologies, and from them retained and blended aspects that proved effective for us, adapted those aspects which proved less effective, and where necessary supplemented them with additional techniques that proved particularly beneficial when tackling petrification. The result - Productive Migration - revealed throughout this and the next few chapters, is an agile methodology specifically geared towards encouraging a productive and motivated migration team that learns-as-it-goes how to solve legacy system petrification.

As we shall see, Productive Migration uses the backlog of currently resisted business value as the driving force behind migration. Specifically, we negotiate with the customer to prioritise the business value in the backlog, and take a few of the top-priority items as the scope of the next increment of migration work. Throughout an increment, we continually evaluate progress, and re-steer that increment as necessary along the way. In later chapters, we will see that the increments themselves are broken

[30] Indeed, our early email correspondence with one agile methodologist (Alistair Cockburn) revealed that he considered legacy systems to be "unstoppable" and agile methodologies to have little relevance to them. After a series of further emails and face-to-face discussion, though, Cockburn has been convinced of the effectiveness of (an early draft of) the approach described here (i.e. Productive Migration), proclaiming: "You have broken through all my pre-existing conceptual barriers" [Cockburn, 2000b].

down into progressively smaller steps, each of which negotiates and elaborates its scope, peels back and legitimises the petrified architecture to accommodate the necessary points of commonality and variability, and then slowly injects support for fragments of the resisted business value into that architecture.

After each increment, we deliver back into customer hands a release of the legacy system which is less petrified than the preceding release with respect to the previously resisted business value which formed that increment's scope. Through a series of such incremental releases, then, we progressively reduce the size of the backlog until we have a system that is considered by the customer to be sufficiently adaptable again, at least with respect to the business value that it has resisted thus far.

Of course, once we have successfully tackled the backlog, the work doesn't stop there; we need to ensure that the system remains adaptable in response to newly emerging requirements change. Consequently, in chapter 12 we will describe a process of Productive Evolution, which carries on the practices of Productive Migration, but with a focus on new requirements rather than backlogged ones.

8.2. High Productivity

Nowadays, the focus at EDP is very much on high productivity. There is a strong commercial imperative to "get things out of the door and into customer hands" as quickly as possible, despite the fact that requirements are usually very uncertain throughout the development process. In fact, a customer is often only clear about the need for a particular requirement at the last minute, when "it was needed yesterday". Long delays between articulation of a late requirement and its fulfilment in code are commercially intolerable.

8.3. Productivity vs. Bureaucracy

To "get things out of the door" quickly we need to shift to a culture of high productivity. Before we can do that, though, we need to understand exactly what productivity is. We have realised that producing and being productive are not the same thing. Not all activity is progress; just because a project is actively producing artefacts does not mean it is progressing towards satisfying business needs. We can think of productivity, then, as a measure of the rate at which business needs are being met. We can define *bureaucracy*, on the other hand, as any production that does not contribute to requirements satisfaction. Figure 8.1 depicts these relationships.

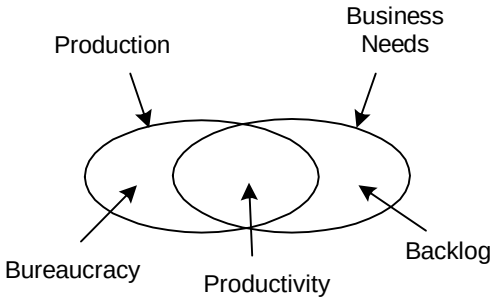

Figure 8.1 – Productivity vs. Bureaucracy

Predictive methodologies, as described in chapter 6, encourage a high level of bureaucratic production, but not necessarily a high level of requirements-satisfying productivity. There is a failure to recognise that not all bulk is muscle[31]. Agile methodologies, on the other hand, strive to eradicate bureaucracy. As we saw in chapter 7, they aim to trim away all the fat, constantly honing themselves to be lean, lightweight, and fighting fit. The focus is one hundred percent on delivering customer value (that is, satisfying business requirements). Anything that detracts from the direct satisfaction of requirements is relegated. Nothing is sacred.

8.4. Legacy Systems and Low Productivity

Legacy systems are likely to emerge in, and ultimately perpetuate, an environment of low productivity, since low productivity allows the pace at which we satisfy requirements to slip further and further behind the rate at which new requirements are emerging. Certainly, the experience at EDP has been that if we are to tackle legacy systems, and prevent them from occurring again in the future, we need to shift into an environment of higher productivity.

8.5. The Sociology of a Productive Team

Throughout this and later chapters, we will talk a great deal about process-oriented aspects of high productivity migration. However, during migration work we have noticed that one of the biggest determining factors for high productivity was team motivation. When motivation was high - when the team was all fired up - there was no stopping them. But once "the glums" set in, progress with migration would come almost to a standstill, and developer focus would shift in and out of various displacement activities such as visiting bookshops, surfing the Internet, and playing with the latest techno-gadgets.

[31] An extreme case of this is the Software Configuration Management Organization [Brown et al. 1999], so caught up in “standards compliance” that they blindly favour paper over progress.

One major advantage of agile techniques may well be their continual bolstering of team spirit by giving continual bursts of success, fostering close bonding with peers, keeping the team on the exciting "edge of chaos", and so on. Nonetheless, there were many occasions at EDP when despite all of this, motivation, and hence productivity, went through the floor. The common factor on those occasions was, unfortunately, management "interference". More specifically, it became clear that non-technical managers would often intervene in ways that they considered harmless or even motivational but actually were completely counter productive. For example, good people were pulled from a migration team to work elsewhere mid-project, with a management promise that "we will replace them with somebody else", as if people were interchangeable Lego pieces. Similarly, a "motivational" talk from management misfired, and was seen by the team as "hitting us with both the stick and the carrot". As a final example, a culture of competitive development (where different teams compete to product the "best" version of some feature) certainly had an initial effect of boosting team energy. Eventually, though, the result was a demoralising feeling that it would have been much more effective if we were all working together rather than against each other.

8.6. Peopleware

Agile methodologies, as published, give little if any assistance for combatting such demoralising factors. Fortunately, mid-project we came across DeMarco and Lister's Peopleware [DeMarco and Lister, 1999], which explores sociological aspects of team productivity. This book was a godsend to the project. Peopleware summarises findings from more than five hundred project histories. The main finding of the book is that a great many impediments to team productivity are sociological. In particular, many of the commonplace measures taken by (often) well-intentioned management have a devastating effect on team sociology and team productivity. Below, we examine some of the measures both noted in the book and taken by EDP management, and describe their negative impact on the team and how we were able to circumvent them. Note that "circumvention" proved ultimately necessary since despite buying Peopleware for senior management, and continually requesting adoption of some of its ideas in what we considered (usually) polite ways, we were generally regarded as arrogant troublemakers. It appears that convincing non-technical managers to "pamper" developers as a boost to productivity is a tremendous challenge, and one with which we have not succeeded well.

Physical Environment EDP is headquartered in a grand stately home, with fine furniture, plush carpets, chandeliers, beautiful gardens, and so on. As charming as the setting is, it does not make the best environment for software development. For example, we were not allowed to stick paper on the walls charting our progress (it might mark the silk wallpaper). We were not allowed bookshelves to hold our (frequently referenced) technical books (they might lean against the wall, and in order to match the other furniture they would be very expensive indeed, so we were told to archive the books in boxes!). As another example, we were unable to use much of the space in the development lab, since a huge antique table with seating for fourteen people dominated the room. All in all, the setting was certainly splendid, but we were powerless to personalise our workspace to accommodate our needs. We were, however, able to circumvent this problem partially. Firstly, we placed old Formica-topped tables around the edges of the room. We weren't afraid to use and abuse these, and could work without the constant worry of "chipping" the fine wood on our official desks. Secondly, we turned the large antique table into a horizontal wall, by covering it with flipcharts to scrawl on, thus making use of the space it occupied and compensating for the lack of usable wall-space. Thirdly, we simply bought cheap bookcases at our own expense and hid them in a little-used room, away from official and disapproving view. A number of these measures certainly infuriated the furniture police (who nagged us constantly), but ultimately meant we had an environment we could work in more effectively.

Overtime Management continually demanded longer and longer hours. It reached the point where some developers were working more than ninety hours per week. This boosted productivity for two to three weeks, but then productivity fell off dramatically. The problem, as pointed out in Peopleware, is that overtime does not work in the long run. Team members are too tired to work effectively, their personal lives suffer, motivation drops, and overtime is always ultimately compensated for by stealthy "undertime" anyway. The migration team eventually realised that overtime was not improving productivity at all, and so effectively banned it themselves. At about five thirty each evening, a senior developer now shouts "Go Home" and the team does just that. On occasion, somebody may stay later to "finish off" something they were working on, but on the whole hours have gone down, the team feels healthier, and productivity has (if anything) gone up as a result.

Ridiculous Deadlines As with many companies, EDP is frequently reactive rather than proactive, meaning that projects are usually started late in the day. As noted at the top of this chapter, this is often because customers are only screaming for something when

they need it immediately. Unfortunately, the management tendency is then to impose a ridiculously impossible deadline on developers in a futile attempt to prevent prospective immediate sales from slipping away. As Peopleware points out, and as confirmed by our migration experience, impossible deadlines have a devastating effect on developer energy levels. Developers know from the outset that they will inevitably be "late" on such projects, and hence it is impossible to remain motivated when "failure" is the only possible outcome. Recognising this, the developers at EDP have now conspired to be permanently honest with managers about what is a realistic deadline for migration work. This has not gone down well with management, but has prevented developers living a lie. In fact, studies described in Peopleware show that the most productive projects are those with no schedule pressure whatsoever. Since our overriding filter is to boost productivity, developers now essentially ignore arbitrary migration deadlines at EDP, and simply commit to being as productive as they can be. Emphasising prioritisation by customers, according to business value for concerns to be addressed in a series of migration increments, ensures that the most valuable work is delivered first and as quickly as possible. If an absolute deadline does exists, and an increment is simply too large to achieve it realistically, then we renegotiate with the customer to reduce the scope of that increment as necessary.

Warm Bodies In an earlier chapter, it was noted that some developers outshine all others. Management, though, continues to view people as interchangeable components. This has resulted in the terrible loss of great developers from the migration team, and frequently their substitution with mediocre "replacements". Pulling good people from a jelled team had a terrible impact on team sociology; motivation dropped and panic set in. Things worsened when we were told who their replacements were going to be. The only tactic we have been able to employ is to say to management "you could give us [some specific person] but we don't know if the project can take the three month slippage it would take to get them up to speed". In parallel, we have then underhandedly approached developers internally that we know to be good, and attempted to coax them to join the team without management approval. We haven't been entirely successful. This is a great pity, because it prevents us from following the ultimate lightweight, high-productivity methodology, as advocated in Peopleware: "get the right people; make them happy so they don't want to leave; turn them loose".

8.7. Quick Tour of Productive Migration

An important lesson we have learned, then, is that only when we have done all we can to boost team productivity by addressing environmental issues and deleterious interference, should turn our attention to process (i.e. methodological) issues.

In the remainder of this chapter and continued in later chapters we describe Productive Migration, which supports the emergence of a highly productive migration team. Agile methodologies focus upon remaining agile amid, and staying on top of, newly emerging and continually evolving requirements. Productive Migration helps us to get on top of the business-value-laden requirements that have *already* overtaken us, and particularly focuses upon tackling the petrification that prevent their support.

8.8. Bare Bones

Productive Migration borrows heavily from other agile methodologies described in the preceding chapter. It is ultra-lightweight, since experience at EDP with heavyweight methodologies has confirmed Cockburn's finding [Cockburn, 2001a] that they are generally resisted, faked, or rejected by developers. It has taken EDP many years to realise that a methodology that is not used has no use. To support effective legacy system migration, then, we must offer a methodology light enough to be embraced yet effective enough to be worthwhile. Consequently Productive Migration in action involves a continual reflection upon whether or not it can be trimmed down further and further yet still remain useful.

Productive Migration as described here has taken about two and a half years to mature “in the trenches”, during which it has been pared down to "the bare bones". If there is any fat, we didn't find it. However, other projects may well need to adapt the methodology to their own appropriate bare bones skeleton, and the methodology itself provides support for this.

8.9. Keeping Pace With Requirements

Productive Migration focuses upon unravelling precisely why a given legacy system resists support for its backlogged requirements (see Figure 8.2), and then progressively loosening that resistance to enable support for those requirements. When we have caught up with the backlog of requirements, we carry this process forward to embrace new requirements as they emerge, in a process termed Productive Evolution (described in a chapter 12). Productive Migration and Productive Evolution embrace practices that help us to boost productivity, and thus prevent requirements charging ahead of us. By

keeping up with the pace of requirements change, we can tackle the systems that are petrified now, and prevent their re-petrification back into legacy systems in the future.

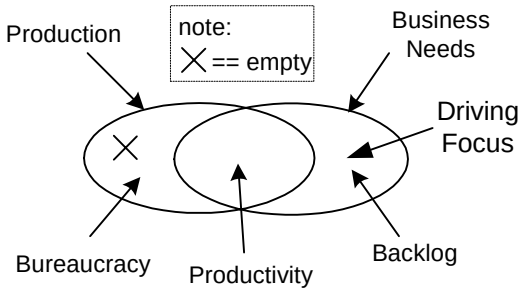

Figure 8.2 – Productive Evolution Focuses on Requirements Backlog

Productive Migration, then, is driven by the backlog of resisted requirements. Since a legacy system is petrified in terms of resisting support for its backlogged requirements, Productive Migration in action focuses upon progressively loosening a legacy system's resistance to those requirements and injecting support for those requirements into the system.

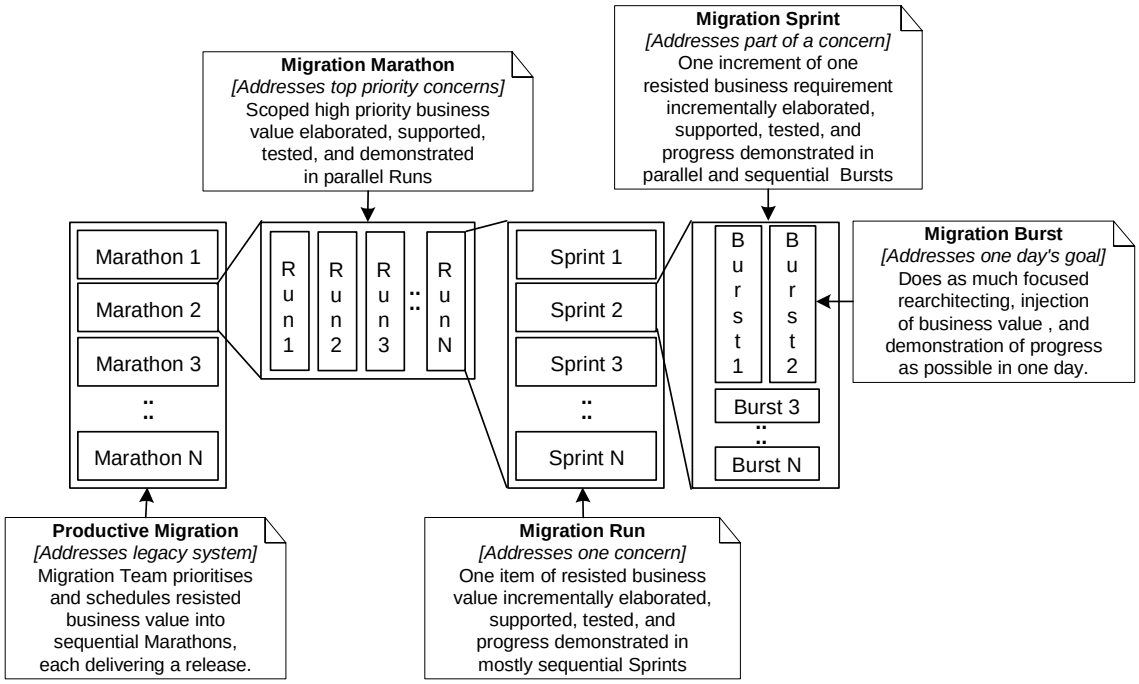

Figure 8.3 – Productive Migration: Marathons, Runs, Sprints, and Bursts

As shown in Figure 8.3, Productive Migration takes an incremental approach to migration, only addressing part of the backlog at a time. Thus, the Migration Team continually prioritises the backlog in terms of business value and always scopes several of the highest priority items into the requirements for the next increment, termed a Migration Marathon. Each successful Migration Marathon will deliver a new release of the legacy system, but with support for the previously resisted requirements for which it

was scoped. To completely tackle the (continually changing and continually reprioritised) backlog, then, there will need to be a sequence of Migration Marathons, each tackling the currently highest priority resisted requirements from the backlog. Chapter 9 describes Migration Marathons in greater detail.

To achieve maximum parallelism, and separation of concerns, a Marathon Team partitions a Migration Marathon's scope (i.e. the resisted requirements it will address) into parallel Migration Runs. The aim is twofold: by maximising parallelism, we should deliver a Migration Marathon's release as quickly as possible, and by separating concerns, each Run Team can maintain greater focus. Since a Marathon includes several distinct items from the backlog, we usually scope one item to each Run. Where there are insufficient Run teams available, Runs serialisation may be necessary. More usually, though, we have found it preferable to say "this Marathon is too big" and reduce its scope accordingly, returning its lowest priority resisted requirements to the top of the backlog for a future Marathon to focus upon. During a Run, customers and developers "flesh out" the requirements for the resisted business value they will address, and (generally) write a body of tests that encapsulate those requirements. Developers take these tests as a specification for the code changes they need to make, and successful execution of the tests (along with hands-on demonstrations) is taken as evidence that the Run has succeeded. Chapter 10 describes Runs in greater detail.

A typical Run may last a couple of months. To manage complexity, and enable demonstration of ongoing progress, a Run Team generally sequences their Run into a number of Sprints (each lasting a week or two). Sprints are where the actual migration work is done – in terms of progressively re-architecting the legacy system to allow the injection of support for the resisted requirement upon which the Run is focused. Sprints home in on the "nitty-gritty" of the Run's requirement. Thus, Sprints "chip away" at a backlogged requirement, fleshing it out and injecting its support incrementally. Sprints, then, enable mid-Run monitoring or progress, and re-steering if things appear to be going off course. Chapter 11 describes Sprints more fully.

Having seen that a Sprint usually lasts a week or two, we can, and often do, break a Sprint down even further, into day-long internal Bursts, wherein the focus is upon a high-energy flurry of activity with a pair of developers striving to make some demonstrable progress each and every day. At the Burst level, developers tend to write Regression Tests (to see if they are doing harm) and Progression Tests (to show that they are doing good). During each Burst, the legacy system is re-architected "just

enough" to enable the legitimate (rather than hacked) injection of a small and manageable fragment of the missing requirement upon which we are focused. Bursts lead to clear daily goals, and a sense of satisfactory progress at the end of each and every day. Bursts are described in detail (along with Sprints) in chapter 11.

8.10. Cultivating Expertise

An important lesson from Productive Migration is that petrification has many causes, and thus we cannot predict in advance what causes will underlie the petrification of a given legacy system. A major goal, then, is to continually cultivate as much migration expertise as possible, learning what works and what doesn't as we tackle the specific and emerging causes of petrification underlying individual legacy systems. As the team gains experience understanding various causes of petrification we don't want to lose that expertise, and hence we strive to capture that expertise in evolving catalogues of recurring themes underpinning petrification, termed Petrifying Patterns. The migration team (including the marathon, run, sprint, and burst teams within it) continually cultivates, in response, novel (and hopefully reusable) patterns of migration expertise (Productive Patterns) with which to address those concerns now and in the future.

At all levels of Productive Migration, then, there is a strong emphasis upon identifying and capturing specific causes of and antidotes to the petrification within each legacy system we tackle. We continually tailor that expertise to the unfolding needs of each migration team and project. In other words, the aim of Productive Migration is not just to migrate a legacy system to an adaptable system. Rather, as a migration team becomes "fitter" throughout their migration work, we also want to capture their evolving expertise in cultivated pattern catalogues. Such pattern catalogues feed-forward into and help guide subsequent migration work, keeping that migration work on the straight and narrow. Chapter 13 elaborates the concepts of Petrifying and Productive Patterns, focusing particularly upon the care that must be exercised in their cultivation and deployment. The appendices contain substantial portions of the particular Petrifying and Productive Pattern catalogues mined whilst addressing legacy systems at EDP.

As noted at the top of this chapter, we have not seen published elsewhere any emphasis on steering agile methodologies toward tackling petrification[32], and more specifically

[32] Having said that, late in the preparation of this thesis we did come across a single, and short, paper which hinted at the possible application of XP within the context of legacy systems [van Deursen et al. 2001]. However, the methodological details were extremely sketchy, with the focus of the paper mostly upon proposing the adoption of certain assistive software tools developed previously by the paper's authors. Note that our own take on the relationship between assistive software tools and legacy system migration is explored in chapter 14.

we have not seen any reference to their enrichment with continually cultivated patterns of migration expertise. This approach, therefore, is seen by us as a valuable contribution to knowledge, and hence is seen as the principle contribution of this thesis.

8.11. Work Products

The primary work product throughout Productive Migration (in addition to greater expertise in the heads of team members, and the pattern catalogues they enrich) is migrated shippable code. The principle test of whether or not other work products (artefacts) are valuable is whether or not they contribute to the productivity of migration and delivery of code both in this project and in future projects. Having said that, there used to be periodic pendulum-like swings at EDP between one particular form of secondary work product - diagrammatic domain-oriented models - being "all the rage" and "avoided like the plague". When diagrammatic models were out of favour at EDP it was usually argued by developers that, in effect, it is close to impossible to produce definitive models of requirements and design until a system has been almost completely built, hence early models will quickly become redundant. Consequently, it was argued, it is common practice for developers to produce code stealthily, and materialise models in an underhand post-development subterfuge, pretending that the models really did precede the code. When diagrammatic modelling was in favour at EDP, on the other hand, it was frequently argued that members of the development team needed to communicate at a suitably high level of abstraction (i.e. above code) and that domain models facilitated this. Furthermore, modelling advocates generally viewed archival of modelling artefacts as essential to enable subsequent generations of developers to “get up to speed” on the same project.

8.12. Speculative Communicative Modelling

Eventually, this oscillation stopped after reflection upon Cockburn's observation, outlined in the previous chapter, that markers have many uses (to remind, inform, and incite). This was an important revelation, and brought to an end (at least for us) a seemingly endless debate (or, less euphemistically, an endless bickering) that ultimately proved futile. Over time it became evident that there are (at least) two distinct forms of modelling, both equally valid, and both with fundamentally different purposes. The first form of modelling, which we may term *speculative communicative* modelling, assumes that models have value beyond the moment of construction, and thus should be retained to accommodate subsequent needs for examination. Models here are seen as capturing knowledge that can be usefully communicated, potentially across both space (diverse

projects) and across time (different generations of a given project). The assumption is that the primary purpose of a model is to capture and communicate knowledge to a future unknown audience even in the absence of the original modeller.

8.13. Domain Explorative Modelling

The second form of modelling, which we may term *domain explorative* modelling, views models mostly as ephemera acting as rapidly evolving short-lived thinking tools of use only when exploring a given domain. The process becomes one of *thinking* with models rather than of *recording* with them. Such “back of an envelope” models are inherently disposable, since their only purpose is to help modellers negotiate shared mental models of the domain under study. Such models are highly personal, they simply jog the participating modellers' memories with respect to: which concerns have already been addressed and negotiated satisfactorily, which concerns have yet to be addressed, and which are purposefully excluded. Since participation in the process of constructing domain explorative models is essential to appreciate them, they are seen as having no particular communicative value outside the team constructing them.

8.14. Agreed Purpose

Recognising that different forms of modelling have different purposes, it soon became clear that it is futile to pursue debates about whether or not modelling is intrinsically "good". Consequently, in Productive Migration (and its follow-on process, Productive Evolution), when dealing with uncertainty in a domain and disagreement with respect to a domain ontology, modelling necessarily follows the domain explorative style. Models capturing expertise of problem and solution domains that are already well understood, and sharing of that expertise beyond the original modellers, suggests the speculative communicative style of modelling. The issue, then, is not whether modelling is speculative communicative or domain explorative. Rather, we now recognise both domain explorative and speculative communicative modelling as valuable and complementary. The critical issue is that everybody associated with a particular modelling effort needs to agree upon and make clear which type of modelling is being performed (and why), in order to avert the danger of that modelling effort being inappropriately labelled bureaucratic or maverick.

8.15. Patterns of Migration Expertise

The principal form of long-living marker constructed during Productive Migration is (as mentioned above) an evolving catalogue of patterns (reusable expertise) emerging from

ongoing migration work. Patterns are discussed in depth in chapter 13, and a collection of migration-oriented patterns uncovered at EDP is catalogued in the appendices.

Patterns have been described as "food for the brain" [Vlissides, 1997b] and hence their principal repository will be the mental models of the project team itself. However, other forms of pattern marker (particularly recorded documentation) can have value both to jog memories and also to communicate the encapsulated expertise to those outside the team that discovered them. The patterns uncovered during Productive Migration (see appendices) encapsulate team learning with respect to legacy system migration. We don't want to lose that experience when the current project ends. Documented patterns can help us to share that knowledge beyond the current project and beyond the current project team. For the patterns catalogued in the appendices, much of that documentation is textual, but where appropriate it is supported by illustrative speculative communicative diagrammatic models. When a novice is later unravelling a documented pattern, on the other hand, personal and ephemeral domain-explorative models may assist that unravelling.

8.16. Domain-Explorative Props

Besides patterns, other markers in Productive Migration are viewed primarily as domain-explorative "props" (as in Crystal) wherein their use and form is entirely up to the project team. The only guidance provided by Productive Migration is that prop markers should inspire the formation of petrification-busting ideas, and facilitate the sharing of them. For example, we have appealed to whiteboards for immediate thinking with ephemeral models, to flipcharts for recording reminders from day to day, and to miniature cloakroom tickets to enumerate and name migration activities showing which are active, which are complete, and which are "on the back burner". In addition, an Intranet-based FAQ, accessible to and modifiable by all team members, records longer-term team-memory “jogs” and thus helps accumulate valuable knowledge as the project progresses.

8.17. Breaking out of The Loop

Productive Migration follows an adaptation of ASD’s "Speculate-Collaborate-Learn" lifecycle [Highsmith, 1999b]. Throughout Productive Migration, the team repeatedly steps back from the legacy system, speculates upon how to resolve the currently highest priority aspect of petrification, collaborates in that resolution, and learns from the experience a (potentially new) pattern of expertise for tackling system petrification in

the future. This “loop” is then repeated for the next aspect of petrification upon which the team must reflect.

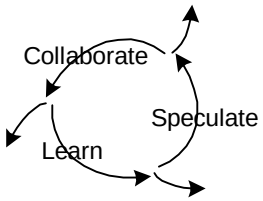

Figure 8.4 – The Loop

The “loop” is only a guideline, not a mandate. As implied in Figure 8.4, above, the team is free to break out of the loop at any time. We have found this both appropriate and beneficial whenever we have needed to explore novel and risky ideas with which to gain fresh insights into tackling particularly tricky aspects of petrification[33]. Indeed, we learn from Highsmith, and from agile methodologies in general, that "creating change-tolerant software - that is, software that is maintainable, enhanceable, and modifiable - … requires a change-tolerant development process." [Highsmith, 1999b]. In particular - and this is deeply important for tackling and eventually preventing legacy system petrification - "to survive in complex environments … [we] must learn to adapt *at least as rapidly* as the environment itself is changing [and] traditional static life cycles are not adaptive enough [to do that].” [ibid.]

Productive Migration, then, empowers the development team to adapt the migration process as the unfolding needs of the project require[34]. In particular, Productive Migration recognises that the high level of creative spark and emergent order necessary to succeed with migration requires sensitivity to, appeal to, and nurturing of the collective talents, inclinations, and motivations of the individuals forming the migration team against the shared contextual backdrop of their migration project. By embracing collective individuality, and encouraging team reflection, learning, and adaptation, Productive Migration strives to cultivate an "inner fire" that enriches and matures the team throughout the process, leaving them healthier at the end of the project than at the beginning. It certainly seems to have done this at EDP.

---

[33] Work described in [Mitleton-Kelly and Papaefthimiou, 2001] supports our experience, showing how this type of approach allows invaluable “gurus” to emerge from within the ranks out of sheer necessity. Such emergence is seen as something that no amount of prescriptive guidance can foster [ibid.].

[34] James Bach argues that successful process evolution/improvement is inherently opportunistic and *must*, therefore, be deeply situated within, and hence part of, individual projects [Bach, 1998]. Attempting process improvement as a separate abstract exercise, on the other hand, often leads to “a phenomenon called *goal displacement*, where people are more likely to fall into the trap of thinking of practices as inherently good or bad, rather than thinking of them as good or bad relative to a particular situation.” [ibid.]

8.18. Review

At the end of the day, Productive Migration is about reducing three types of risk: The first risk is that of complexity – by adopting a piecemeal approach to migration, Productive Migration breaks a potentially overwhelming task down into manageable increments. The second risk is that of scope – by focusing on currently resisted business value, rather than purely speculative concerns, Productive Migration ensures that we are addressing the specific aspects of petrification that have a real and detrimental effect on the business. The third risk relates to shared understanding – by continually negotiating and elaborating requirements, and continually monitoring and discussing progress, with re-steering mechanisms should things go wrong, Productive Migration promotes a mutual understanding about, and shared commitment to, the direction in which migration is headed, thus significantly reducing the likelihood of late and possibly nasty surprises.

In the next chapter, we elaborate Migration Marathons, which are the highest-granularity increments in Productive Migration. Later chapters will focus upon the progressively lower-granularity increments (Runs, Sprints, and Bursts) from which Migration Marathons are ultimately composed. Note that throughout these increments we need to give a migration team sufficient freedom to fulfil their migration work in whatever way is appropriate to their emerging needs, without deleterious outside interference and impediments. Ultimately, then, in Productive Migration nothing is sacred other than getting the job done.

Chapter 9

Migration Marathons

Stones dissolved to water do convert.

- William Shakespeare, *The Rape of Lucrece* Stanza 85

9.1. Preview

In this chapter we elaborate how Productive Migration manages the sheer size of legacy system migration by addressing it incrementally. Each increment is termed a Migration Marathon[35]. Each Migration Marathon is scoped according to both the urgency of outstanding (resisted) business value and the readiness of developers to accommodate that business value. Ongoing feedback via frequent demonstrations of progress and discussion of issues arising enables just-in-time Marathon steering. Continual steering keeps a Marathon on course (even when the course itself changes!) and ensures a satisfactory release is delivered into customer hands.

9.2. Incremental Migration

One of the greatest obstacles to migration is the sheer size of the problem, since (as we saw in chapter 1) legacy systems tend to be large, and the causes of petrification tend to be many. Legacy systems, though, have a tremendous advantage over green-field developments; they are already installed and in-use. With a green-field development, users have to wait until the system is largely finished before they can begin to use it effectively. This might seem obvious, but it has an important implication: Since a legacy system is already usable, we don't have to wait for its complete migration to an adaptable system before we can use it again. Thus, Productive Migration performs

[35] Note that throughout Productive Migration (and Productive Evolution) we employ a Running metaphor, including Marathons, Runs, Sprints, Bursts, and so on. We found this metaphor quite useful, particularly since a number of the developers we worked with were experienced cross-country runners. Other metaphors, of course, may be equally valid, or in some contexts even more appealing. For example, we toyed with using a Sailing metaphor, wherein the destination is not clear, landmarks on the horizon give clues about where we are, unexpected winds can blow us off course, requiring all hands to the decks, etc. Similarly, Stuart Kent found a Cycling metaphor more effective, particularly multi-stage competitions such as (quoting from a personal conversation) “the Tour-de-France, wherein each cycling team has members that help each other finish the race. Team members keep their ears and eyes open to potential risks, changing situations and so on. Detailed tactics will have to be invented and strategy adapted on the fly as the race progresses. The race has various stages, with rest days; some stages are difficult (mountain climbs and the like, where the team may have specialists), others are speed trials (just brute force speed - little tactics), and so on.” Thus, although we found the Running metaphor useful ourselves, we recognise that it may benefit other teams to drop that analogy if they find it unappealing, and adopt another metaphor with which they feel more comfortable.

legacy system migration incrementally. Rather than attempting an overly ambitious wholesale, one-shot rewrite of a legacy system, which (as discussed in chapter 3) will likely fail in the long run, we start with an existing legacy system base, address its petrification one step at a time, and continually monitor progress throughout migration to determine what to do next. There is no big up-front plan, other than in very general terms, rather we think in terms of an unfolding series of short sub-projects (increments), each learning from those that precede it[36].

We have witnessed three advantages of incremental, migration: firstly, since we can tackle petrification one step at a time, we can deploy the system under migration back in to the hands of users at frequent intervals, and get valuable feedback along the way about whether or not we are succeeding. Secondly, since users can still continue to work with the system throughout its migration, there should be some relief from the time pressures that affect many green-field projects. We have found that this gives us time to learn as we go along, to reprioritise our migration activities as we need to, and not be lured by futile "quick fixes" (Silver Bullets [Brown et al. 1999]), which tempt those facing deadline panic. Thirdly, the breathing space between spurts of migration can be scheduled for the addition of high-priority newly emerging functionality (see chapter 12) that cannot wait until all migration is complete.

9.3. Marathon Pacing

In essence, adopting an incremental approach allows us to manage the often-overwhelming size and complexity of legacy system migration. Thus, as depicted in Figure 9.1, Productive Migration follows a series of paced medium sized projects (typically lasting up to a couple of months each), which we term *Migration Marathons*, each of which tackles only some of the aspects of petrification. The intention is that, after each Marathon, the system is less petrified than it was before that Marathon began, and the progressively more adaptable system is deployed back into customer hands in place of the original legacy system. This process of partial migration followed by redeployment continues through a succession of Marathons, possibly with a breathing space between them, until the system is considered sufficiently adaptable again with respect to its known resisted requirements.

[36] We are not alone here: As we saw in chapter 3, Brodie and Stonebraker also [Brodie and Stonebraker, 1995] advocate an incremental approach. Furthermore, Pigoski asserts that “One of the most important practices for a maintenance organization is to have regularly scheduled releases”, recounting that “Successful maintainers attribute much of their success to [such] scheduled releases.” [Pigoski, 1997]

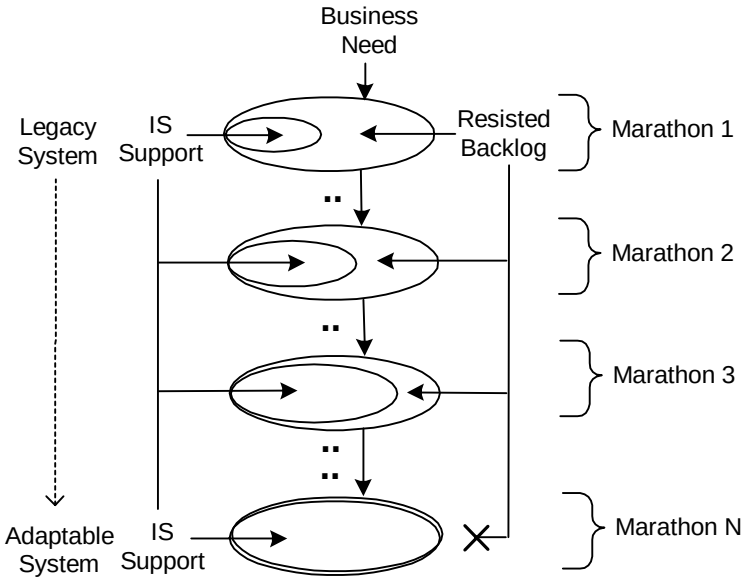

Figure 9.1 – Migration Marathon Pacing

9.4. Marathon Scoping

Migration Marathons are deliberately high granularity, each usually spanning at least a couple of months and encompassing enough high priority resisted requirements to constitute a worthwhile release. In our own migration work, we have seen Marathons give developers sufficient project scope to think at a more abstract level about where the project is headed, and nudge it gently when things appear to be going in the wrong direction. In contrast, we found that an endless series of mad-dashes (e.g. individual Sprints, as in the Scrum methodology) can be just as overwhelming as the monolithic approach, since developers tend to become swamped by a mass of "bitty" projects, unable to "see the wood for the trees".

Before a Marathon begins, a preliminary *Marathon Scoping* session hashes out the general areas of petrification to be tackled. As we see in Figure 9.2, the migration team collaborates with project sponsors and other management and external stakeholders, to negotiate a very general “lay of the land”, forming a requirements *brief* (in the sense familiar to buildings architects and other designers [Lawson, 1997]) rather than an unwavering detailed mandate. The focus is very much upon determining which areas of petrification to tackle and the formation of general strategies with which to tackle them. Tactics are deliberately omitted from the negotiation, since they will be allowed to emerge "just in time" throughout the Marathon itself (during further Marathon Scoping sessions, and throughout the Runs, Sprints, and Bursts that comprise the Marathon),

when the characteristics of the particular areas of petrification being tackled are better understood.

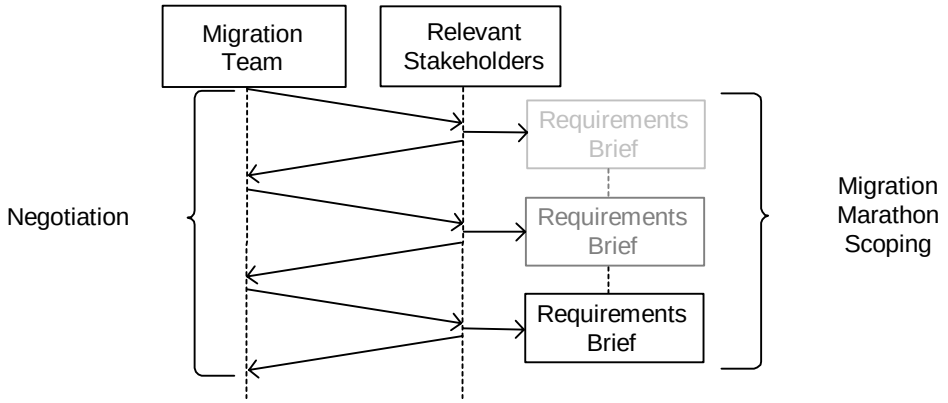

Figure 9.2 – Preliminary Migration Marathon Scoping

9.5. Focus on Resisted Business Value

How do we decide which areas of petrification to address during each Marathon? This is an area where we differ substantially from current approaches to legacy system migration, such as [Brodie and Stonebraker, 1995]. Simply picking an area of petrification that looks particularly exciting technically (leaving the boring work until last, as we have seen done at EDP in the past) does not guarantee that we are addressing the most pressing problems. We eventually learned that a more promising approach was to draw inspiration from XP [Beck and Fowler, 2001], and work with customers to prioritise outstanding work in terms of the business value it delivers.

9.6. Prioritising Backlogged Requirements

If we remember that a legacy system is a system that resists the reflection of business process change, we can collaborate with the customer[37] to enumerate the business process change that the legacy system still continues to resist, resulting in an explicit list of backlogged requirements.

Legacy system maintainers often see backlogged requirements as an unbearable weight on their shoulders from which they can never escape ("the Siberian Salt Mine" as one

[37] Note that we are using the term "customer" here not in the strict sense of somebody who buys goods or services from us, but in the more general XP sense of one or more people authorised to negotiate and make commitments with us on behalf of the business. We need to be careful here, though, to ensure that the customer is a true representative of all ultimate decision-makers within the business, since one early and high profile (and ultimately terminated) application of XP has received fair criticism for not ensuring that the (amusingly homophonic) "Goal Donor" and "Gold Owner" were in agreement [Wiki Community, 2000]. Consequently, Productive Migration has learned (through our own similar early failings) to pay particular attention to ensuring that all relevant stakeholders, or their fully authorised representatives, are present in negotiations.

developer described it)[38]. Productive Migration strives to turn that foe into our ally. As depicted in Figure 9.3, we negotiate with the customer (and, hence, the various other stakeholders they represent) to prioritise (and continually reprioritise) the list of backlogged requirements in terms of the severity of lost business value.

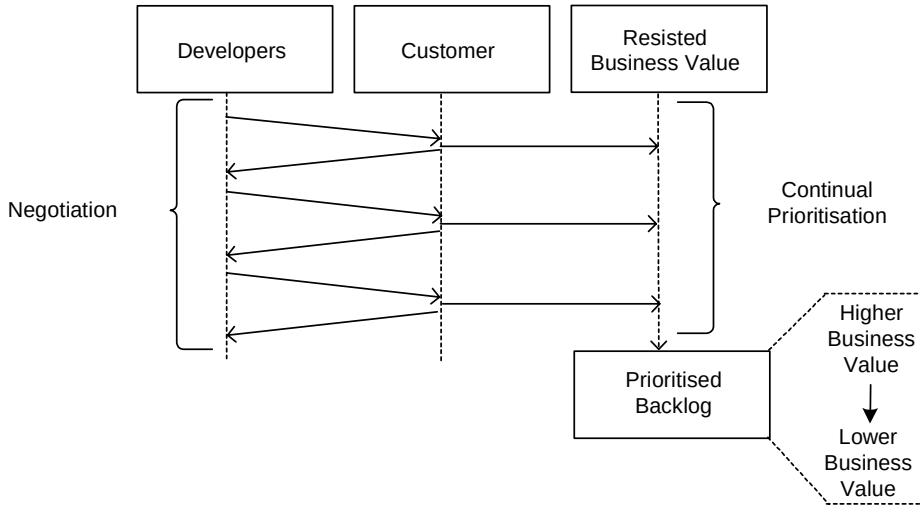

Figure 9.3 – Prioritising the Backlog

As an example, Figure 9.4 depicts a real, albeit partial, prioritised backlog of previously resisted business value, taken from migration work at EDP:

Backlog Prioritised by Business Value

1. Credit Card Payment: payment by credit card
2. Kitchenware Back Ordering: back ordering of kitchenware
3. Flexible Stock Reporting: configuration-file driven stock reporting
4. Multiple Product Images: more than one image per product
5. Product Comparison: side by side product feature comparison
6. Flexible Order Reporting:configuration-file driven order reporting
7. Quotations: fixed price quotations without ordering
8. Paginated Alternates: scrolling through alternates

.. ..
.. ..

Figure 9.4 – Backlog Prioritised by Business Value

9.7. If It Makes Sense to Developers

The XP principle of having the customer prioritise backlogged requirements is clearly better than prioritising on developer whim. Unfortunately, we have found this to be extremely difficult in practice, since frequently the customer considers all outstanding requirements to be high priority! Even where there is a clear priority, we have discovered that there is often a more natural and effective prioritisation from the

[38] In a sense, though, a backlog is an asset rather than a liability. It would perhaps be more worrying if no requirements changes were coming in at all, since this may indicate that nobody was using our system any more. Pigoski and others [Pigoski, 1997], then, view a backlog positively as a healthy order book.

development perspective. In particular, developers can often achieve greater progress in migration by "tackling two birds with one stone" - "If I do this first, it will help me with that later". This leads to a peculiar advantage of having a backlog of resisted requirements. We saw in chapter 7 (surveying agile methodologies, and the concept of YAGNI) that it is ill advised to speculate forward about the architectural commonality and variability needs of unknown future requirements. We have found, though, that with a backlog of requirements for a legacy system, we do not need to be entirely speculative. When we are looking at a particular backlogged requirement, we can also look towards the architectural needs of other requirements on the backlog[39]. By grouping together several requirements with similar architectural (i.e. commonality and variability) needs into the same Marathon, we can ensure that we are successfully tackling the petrification concerns of them all at the same time.

As an example, in Figure 9.4 there is a high-priority need for highly flexible reporting on stock. Much lower priority, in business terms, is the backlogged requirement for flexible reporting of customer order history. Developers may well decide that these requirements have much in common with respect to their architectural needs, and thus negotiate “promotion” of order history reporting, enabling it to be tackled in unison with stock reporting. Remember that Productive Migration strives to be highly productive, if it is more productive to tackle two missing requirements together rather than separately, then they should be tackled together.

As another example, turning to the backlog in Figure 9.4, Multiple Product Image support could benefit from the forwards-and-backwards scrolling required for Pagination of Alternates, hence it makes sense from a development perspective to group these two requirements together in the same Marathon.

Having considered the developer perspective, then, negotiations with the customer result in a reprioritisation of the backlog as shown in Figure 9.5.

Productive Migration, then, sees prioritisation of the backlog of resisted requirements as necessarily a negotiation between both customers and developers. From this negotiation emerges a re-prioritised backlog of previously resisted requirements that will deliver the "biggest bang for the buck".

[39] It took an embarrassingly long time to realise something that is so “obvious” in retrospect. Nevertheless, this revelation has considerably shaped Productive Migration.

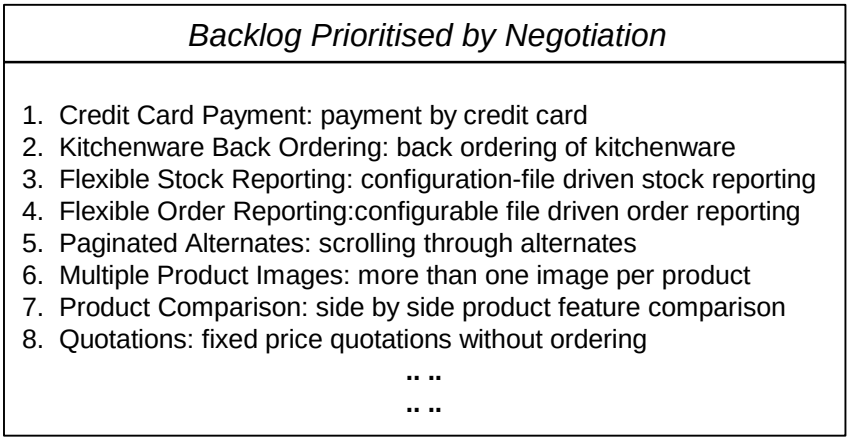

Figure 9.5 – Backlog Prioritised by Negotiation

9.8. Marathon Scope

A small number of the highest priority items on the backlog are taken as the scope of the first Marathon. For example, the customer and developers may negotiate a Marathon comprising the four highest-priority items from the backlog, as depicted in Figure 9.6:

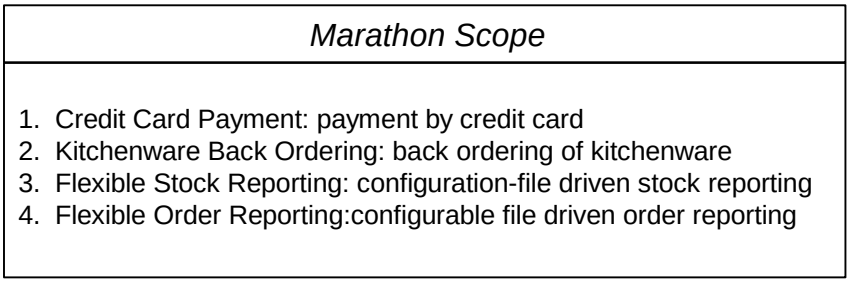

Figure 9.6 – High-Priority Marathon Scope

The next highest priority items will form the preliminary scope of the next Marathon, and so on. Of course, throughout Productive Migration changing circumstances result in a continual reprioritisation of the backlog. Thus, we cannot expect to completely scope out all subsequent Marathons up-front, rather the ultimate scope of a Marathon is generally dependent upon the relative priority of items on the backlog just before that Marathon begins.

During a Marathon we strive to uncover the sources of petrification underlying the legacy system's resistance to the particular business-value-laden requirements we are addressing, and re-architect the system just enough to reflect the appropriate commonality and variability assumptions for those requirements.

This approach of addressing the backlog in business-value order ensures we are always undertaking migration work that delivers the greatest value to the business, and this is important. As Sneed has pointed out, we are “continually confronted with the need to

justify reengineering. The user wants to know what the benefits are" [Sneed, 1995]. Here, the customer will see the benefits for themselves: they will see real value being delivered from each Migration Marathon, rather than faithfully waiting for some abstract technical migration that only promises to be useful. We have seen for ourselves in our own migration work that the early delivery of high business value from the first few Marathons increases the perception that migration work is valuable, rather than simply an invisible drain on resources, and thus helps increase support for further migration work. Furthermore, since a complete migration can be a long drawn out affair - possibly spanning years worth of staged Marathons - we ensure that only low priority items are years away and thus take some pressure and desperation away from the schedule.

9.9. Marathon Demos

Usually throughout a given Marathon, *Marathon Demos*[40] are held wherein successful progress is demonstrated to those outside the immediate migration team. Marathon Demos help to prevent late and nasty surprises, and give just-in-time feedback should the migration team be heading in dubious directions.

We learned the hard way about the value of ongoing Marathon Demos (and other demos, as described in later chapters). We didn't have enough of them early on in our migration work, and this led to considerable embarrassment and misunderstanding when the demos eventually took place; we had significantly misread the customer's shifting requirements and hence were not satisfying some of them very well at all. If you don't know where the requirements and their relative priorities are shifting during migration, you cannot expect to catch up with them.

9.10. Marathon Steering

Throughout fulfilment of a Marathon, the developers and customers comprising the migration team (and occasionally other customers, management, and other stakeholders) meet in frequent (perhaps every few days) and short *Marathon Steering* sessions. The purpose here is to share lessons learned (including potential petrifying and productive

[40] We were inspired by Scrum in the value of demos to receive invaluable feedback, although we probably perform them more often than Scrum, which concentrates demos mainly at the end of Sprints. In particular, there are multiple "levels" of granularity in Productive Migration (Marathons, Runs, Sprints, and Bursts, as revealed in later chapters), rather than the essentially single low-level Sprints of Scrum. Each of our granularity levels attracts participants with potentially different perspectives and expectations. By exposing participants at each level of granularity to ongoing demos suitably focused at that level of granularity, we enable those participants to both evaluate the emerging deliverable at a suitable level of abstraction and share insights which other levels of granularity may miss.

patterns), to air concerns about unsatisfactory demos, clarify Marathon scope, bring to the table necessary changes in scope, re-evaluate estimates of resource requirements, and so on. The outcome of these Marathon Steering sessions is generally a re-negotiation[41] and re-steering of the Marathon to keep it on track.

As an example, one Marathon we participated in originally included a requirement for fixed-price quotations, which had been lingering on the backlog of resisted requirements for at least a year. After struggling with this requirement for some time, developers realised that the organisation currently had no clearly thought out business process support for such quotations. Consequently, this particular aspect of order pricing was renegotiated in the next Marathon steering session for deferral to a later Marathon when the organisation had more time to reflect on its business process implications.

We have found steering sessions to be the lifeblood of a migration project, ensuring that everybody is moving essentially in the same direction. An important phrase that we often use to describe the focus of steering is "continually negotiated goal alignment"; when we are undertaking migration work we are continually collaborating both within the migration team and with interested external stakeholders to ensure that everybody shares compatible evolving mental models about where the project is going.

9.11. Sociological Benefits

In the previous chapter we saw that Scrum meetings – which have certainly influenced the various forms of steering session in Productive Migration - are claimed to promote: a sense of urgency; knowledge sharing; dense communication; and honest delivery of bad news as well as good. Having followed Productive Migration for more than two years, we have found the following additional benefits for ongoing negotiative (i.e. steering) sessions:

- Reduce fear of "sinking" in a sea of uncertainty, by almost magically responding just in time to urgent unknowns. Productive Migration provides just enough buoyancy to keep team members afloat without stifling their freedom of movement.
- Promote a sense of belonging and kinship, eliminating the isolation that stems from the "divide and conquer" inherent in many overly prescriptive methodologies.

[41] We consider this term important, and more precise than more general terms such as “communication” as used by most agile methodologists. In particular, we have generally found that few steering participants have a complete understanding of where things should be headed – since so much is open to emerging conditions. Thus, although steering participants may arrive with (often vague) personal agendas, and loose goals to satisfy them, we have found that effective steering requires participant to buy-in to negotiated goal alignment, rather than simply information exchange or, worse still, grandstanding.

- Promote a sense of self-worth and appreciation, by encouraging and actively listening to team members and responding to whatever holds their productivity back.
- Keep energy levels and motivation high, by expecting, encouraging, and supporting everybody to be as productive as they can be, but no more productive than that.
- Finally, and most importantly, ongoing scoping, steering, and demo sessions ensure that changing requirements are tracked closely and responded to quickly. By continually keeping their eyes and ears open to emerging project risks and opportunities, and continually forming and prioritising the project backlog, the team can be sure that they do not do any work they do not need to and remain completely focused upon the emerging work that they absolutely must do.

9.12. Productivity Filter

Throughout our migration work, ongoing steering sessions certainly have helped keep projects on track and facilitate just-in-time emergence and team learning and socialisation. However, early experience with such sessions has led us to the realisation that they have the potential to descend into motivation-sapping grandstanding and intellectual debate, dragging on and on with few decisions being made. Many steering sessions (particularly Marathon Steering sessions) started to be dreaded by some as "yet more time wasting meetings". We needed something to focus the sessions, and to make them productive again[42]. To achieve this, Productive Migration scoping and steering sessions now tend to employ a very simple high-level "filter" that guides team negotiation. The filter simply involves continually asking: "Will this decision increase or decrease team productivity?" Anything that increases team productivity is almost a "shoo-in", whereas anything that decreases team productivity is most certainly frowned-upon (termed a "boot-out" by one wag we have worked with). Even in places where the answer isn't clear, asking this question brings the important issues back into perspective. We have found that, by employing a simple filter, potentially endless debates during ongoing negotiation are now successfully cut short, ensuring that scoping and steering are again productive; they move along relatively quickly with a high level of energy and with clear-cut decisions along the way.

[42] The only guidance we found from other agile methodologies was the XP insistence that all attendees stand up, to quite literally keep them on their toes. Unfortunately, this didn't work well with, for example, the Managing Director of EDP, who understandably preferred the comfort of a chair.

9.13. Crossing the Finish Line

At the end of a completed Marathon is a final Marathon Demo[43], during which the migration team declares its work done, and the finished deliverable is (hopefully) accepted for deployment by the customer. This delivered product can then be tried "in the field" for some time, with appropriate feedback feeding into the next Marathon at some later date. If, due to poor quality steering, the final result of a Marathon is wholly unsatisfactory, the incremental approach inherent in Productive Migration enables us to rollback to a previously good Marathon, or to take remedial action via an appropriate Compensating Marathon.

When a Marathon has been completed to the satisfaction of the migration team, and where appropriate customer approval is received at a final Marathon Demo, the Marathon is said to *cross the finish line*. This is stating more than "here is a finished working product". It is also a declaration that the system is now more adaptable than it was before the Marathon began. As a succession of Marathons unfolds, and their results are deployed, the system gradually becomes less and less petrified, until eventually the system is considered sufficiently adaptable again.

9.14. It’s a Journey, Not a Destination

Importantly, throughout Productive Migration adaptability must be seen as a journey, not a destination and, hence, the post-Marathon system will require careful nurturing to retain the levels of adaptability that have been injected into it. It is difficult to emphasise strongly enough that if we resort back to our old practices, the system will almost inevitably become completely petrified again.

Thus, once results of the first Marathon have been deployed, we need to bring in practices that retain its injected levels of adaptability, both while other Marathons are unfolding and being deployed, and when all Marathons are eventually done. To keep a mid- and post-migration system adaptable, and prevent it from petrifying again, as ongoing requirements changes impact the areas that completed Marathons have already addressed, we can appeal to the principles of Productive Evolution, which are detailed in a later chapter.

9.15. Review

In summary, as depicted in Figure 9.7, for a Migration Marathon, initial scoping involves negotiating and costing a high-level requirements brief for resisted business

[43] Similar in spirit to the final Sprint demo of Scrum (see chapter 7).

value. Prioritising and scheduling migration work according to both urgency of resisted business value and development efficacy ensures that we deliver the greatest benefit in the most productive way we can realistically achieve. Continually re-steering a Migration Marathon, according to ongoing feedback in response to demonstrated progress, prevents us from stumbling at the last hurdle, and promotes consensus about where we are heading. When we have "crossed the finish line" with a completed Marathon, satisfactorily supporting previously resisted business value, we are free to start scoping out the next Marathon.

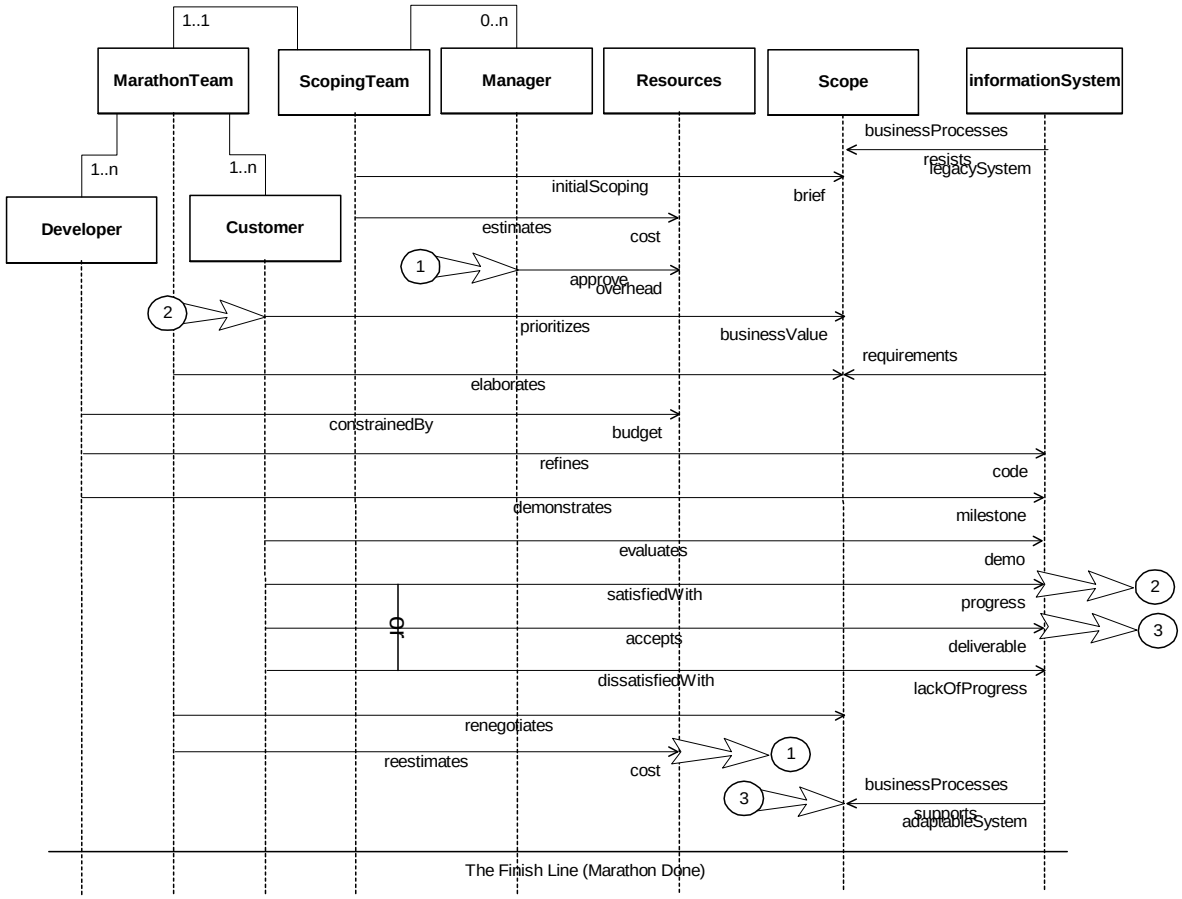

Figure 9.7 – Migration Marathon (Bare Bones)

In the following chapter, we go deeper still into Productive Migration. We will see that by breaking Marathons into constituent Runs - each focused on a separate area of resisted business value - we can maximise parallel work, concentrate individual attention, embrace just-in-time requirements elaboration, and generally keep migration work highly productive and continually on track.

Chapter 10

Migration Runs

T'is time: descend: be stone no more.

- William Shakespeare, *The Winter's Tale* V, 3

10.1. Preview

In the previous chapter, we saw how a Migration Marathon groups together the highest priority items from the backlog of resisted requirements, and ultimately delivers support for them in an end-of-Marathon release. In this chapter, we explore how Migration Marathons are frequently profitably decomposed into constituent Migration Runs. Each area of missing business value scoped within a Marathon generally becomes the focus of a separate Migration Run. The intent is to enable Run teams to work as far as possible in parallel, thus maximising concurrency within a Marathon. Where possible, a Run team works independently, allowing other Run teams to focus upon their own concerns. However, Run teams in trouble are actively encouraged to bring real problems back to the Marathon team as a whole, since unchecked problems in one Run can trickle out to negatively affect the completion of the Marathon as a whole.

10.2. Marathon Running

Completion of a Migration Marathon is the collective responsibility of the whole migration team. However, we have found that it usually takes the attention and drive of focused individuals to uncover the nitty-gritty details that are lost in a high-level overview. Each Marathon team member, then, both contributes to the general picture in terms of understanding and negotiating the Marathon's overall direction (horizontal participation), and becomes deeply immersed in seeing migration through to the very end in a small number of specific areas within that Marathon (vertical participation).

Thus, during Marathon Scoping (and, later, Marathon Steering), having aggregated high-priority resisted business value into the scope of a single release to the customer, we partition the provision of support for that resisted business value into a number of constituent parallel Runs each typically lasting for the duration of the Marathon as a whole (see Figure 10.1). It is the combined deliverables of the Runs within a Marathon that satisfy the scope of the Marathon as a whole.

As depicted in Figure 10.1, to undertake Runs, the Marathon team partitions itself into several Run teams. Just as a Marathon team comprises both developers and customers, so will each Run team. We have found relying on volunteers for each Run team to be more effective than dictating team membership. Throughout our own migration work there has never been a Run short of volunteers, and we have generally found team members themselves to be extremely effective at gathering a good mix of necessary skills and experience appropriate to the needs of each Run.

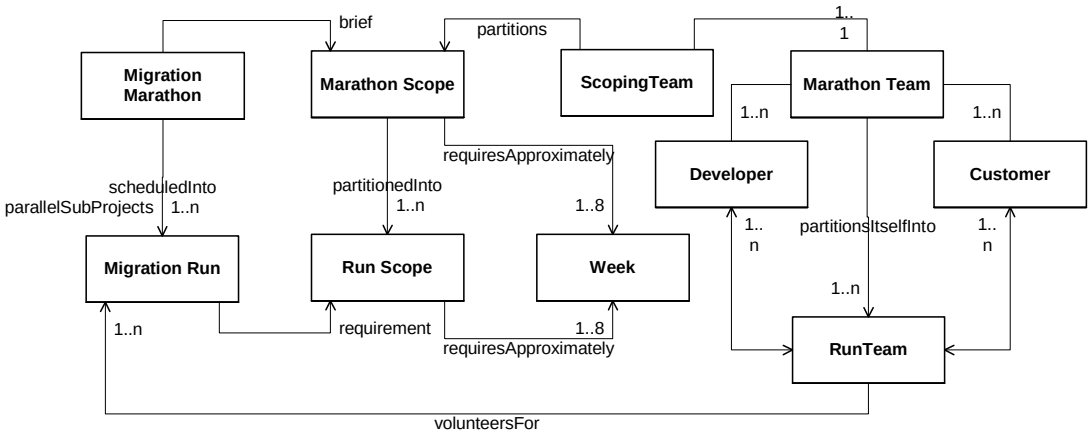

Figure 10.1 – Marathons Are Scheduled Into Runs

10.3. Partitioning Marathon Scope

We have found that Marathon Scoping (see the previous chapter) tends to result in a surprisingly clear understanding of how to partition the scope of a Marathon into separate Runs. Ultimately, each Run is about enabling a legacy system to support some specific backlogged business process change that it previously resisted.

Thus, the Marathon depicted in Figure 10.2 (a duplicate of Figure 9.6) is readily partitioned into the four Runs depicted in Figure 10.3. Sometimes, of course, the partitioning will not be quite as clear cut as this, and it may be necessary to partition an individual item from the Maration scope into several Runs, or combine more than one item into a single Run.

Marathon Scope
1. Credit Card Payment: payment by credit card 2. Kitchenware Back Ordering: back ordering of kitchenware 3. Flexible Stock Reporting: configuration-file driven stock reporting 4. Flexible Order Reporting:configurable file driven order reporting

Figure 10.2 – Marathon Scope

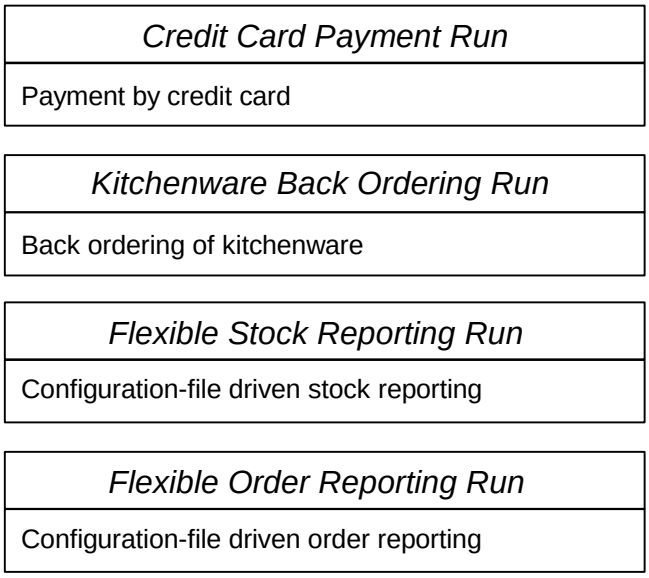

Figure 10.3 – Partitioning Marathon Scope into Runs

Once the general scope of each Run within a Marathon has been determined, the Marathon enters a second phase, termed *Marathon Running*. Marathon Running consists of usually parallel completion of the Runs within the Marathon (coordinated by Marathon Steering Sessions, as noted in the preceding chapter). Note that each Run typically has similar duration to the Marathon as a whole, although where a particular Run finishes early, we have often found that its participants are willing to contribute to other Runs that are still underway.

Looking at the Runs in 10.3, Credit Card Payment and Kitchenware Back Ordering are independent Runs and can be completed in relative isolation. Flexible Stock Reporting and Flexible Order Reporting, on the other hand (as we saw in chapter 9), will almost certainly overlap from a development perspective. For example, they may both be configured via XML, and rendered via a common reporting class. Thus, it makes sense for the Run teams scoped to these two Runs to collaborate quite closely, perhaps even uniting into a single Run team for some time to satisfy common requirements.

10.4. Creative Freedom

At the Marathon level, the scope of each Run is deliberately kept at a high level. Usually, the only external requirement negotiated with a Run team during Marathon Scoping is the (general) scope of their Run in the form of a brief, which outlines the resisted business value they need to support. Thus, for example, the Kitchenware Back Ordering Run in Figure 10.4 has the extremely terse brief to support “Back ordering of kitchenware”. To be useful as a sound basis for development, this brief would clearly need elaborating. A brief, then, is really only a stimulus for further work.

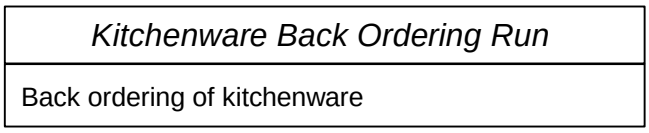

Figure 10.4 – Kitchenware Back Ordering Run Brief

As depicted in Figure 10.5, below, having been chartered with a particular brief, a Run team is free to adopt or cultivate whatever practices helps them to flesh out that brief and complete the Run most effectively; tackling a particular aspect of system petrification is extremely challenging, and experience has taught that considerable creative freedom is frequently required to "crack a hard nut".

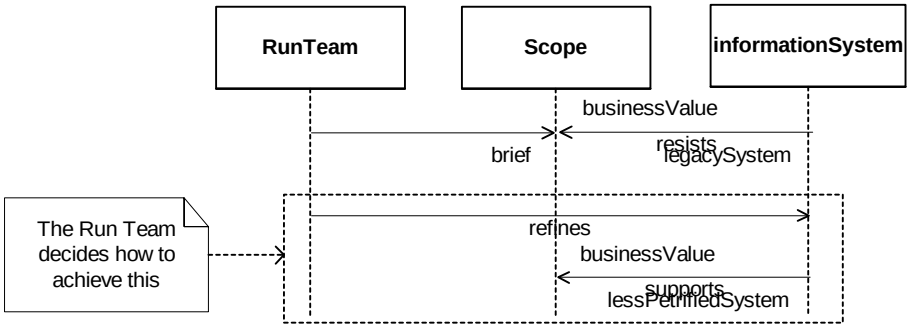

Figure 10.5 – Migration Run Teams Roll Their Own Process

10.5. Requirements Elaboration

Having said that a Run team is free to use whatever approach they choose to complete their Run, most Run teams at EDP (although not all) now embrace the principle from other agile methodologies wherein developers pursue an ongoing dialogue with the customer. Through such dialogue the customer and the developers unravel detailed requirements for the previously resisted business value that they wish to accommodate.

10.6. Run Steering

Each Run team tends to meet frequently (typically daily), in a series of *Run Steering Sessions*, to continually flesh out the requirements for the Run[44]. It is during these sessions that progress thus far is discussed, and Acceptance Tests are elaborated. Acceptance Tests (described in greater detail below) are tests that will only succeed if the previously resisted business value scoped for the Run is successfully incorporated into the system.

[44] By meeting frequently, and discussing emerging concerns, we have found that we reduce the likelihood of Requirement Jeopardy [Brown et al. 1999], wherein important, subtle, or changing requirements are missed.

In addition to developers and customers elaborating requirements, in our experience there can also be a trimming down of requirements. We have found that this generally happens when developers discover that implementing some sub-feature of the Run's scope, or even the Run as a whole, is much more difficult than originally anticipated. It may then be sensible to reconsider (in the next Marathon Steering session) whether or not to defer or eliminate that sub-feature, so that the remainder of the Run can progress productively, or to defer or eliminate the complete Run so that the Marathon as a whole can progress productively. We have been surprised time and time again by how many initially "essential" features are subsequently deferrable in order to get software that is "good enough" out of the door and into customer hands. As an example, during a migration project we worked on, the acceptance of Credit Card Payments (a Run depicted in Figure 10.3) was initially seen to be one of the highest-priority resisted requirements for a certain system. Unfortunately, after reflection by development, it transpired that credit card support would require months of development work in collaboration with third party specialists. Furthermore, that work could not even begin other petrified aspects of goods payment were thought out and addressed beforehand. Consequently, Credit Card Payment was reprioritised, and became one of the very last things to be implemented and released (in a separate Run, in a later Marathon).

10.7. Run Demos

As with Marathons, throughout a Run, teams typically demonstrate their ongoing progress via periodic *Run Demos* (usually when some significant milestone is reached). This usually involves both execution of the software product under development, and review of other significant artefacts (such as designs, requirements, documentation, and so on that the Run teams wishes to share[45]). For the Kitchenware Back Ordering Run mentioned above, then, an early Run Demo may involve demonstrating a back order being entered, with a dialog box acknowledging the back order, but without any real

[45] We rely on responsibility here, assuming that the Run team will not want to "hide" things that other people may need to see. In particular, we do not force Run teams to share anything other than the deliverable code, since we have found that, for example, enforced dragging out preliminary design documents for public scrutiny is often uncomfortable for developers, and is seen by many Run teams as unwarranted external "meddling" in matters that only the Run team itself is sufficiently immersed in to fully understand. We rely instead on the professionalism of the Run team to bring to the table any hidden concerns that need to be shared with others. Having customers in the Run team, we have found, helps prevent developers "conspiring" to hide bad news, as does familiarity with other Run teams sharing "warts and all" during their own demos and steering sessions. The most important lesson we have learned ourselves is that Run teams are generally willing to share both the good and bad news with respect to their progress, so long as we never "kill the messenger". On rare occasion, of course, this "honour system" has backfired badly and certain Run teams haven't delivered at the end of the day. In such cases, we have usually reallocated their scope to another Run, and ensured that a more experienced developer is mixed in with that particular Run team in future.

back end support at this stage. As the Run progresses, additional Run Demos will reveal fuller support in the back end application.

We have found that Run Demos help with early identification of a Run that is going completely off-track, so that its early redirection can be renegotiated (if necessary, with the rest of the Marathon team) either immediately, or in the next steering session. Without such review, a project is in danger of Failure to Audit [Brown et al. 1999], potentially leading to developer complacency, customer dissatisfaction, and consequent cost and time overruns due to post delivery scrambles to "fix" the product.

10.8. Run Done

Again, as with a Marathon, a Run team generally concludes a Run with a final Run Demo. In addition to demonstration of the deliverable, a Run Demo invariably includes the customer successfully running any Acceptance Tests for the Run. Our experience has been that there are rarely any nasty surprises here, since ongoing steering sessions and Run Demos have typically kept everybody well aware of ongoing progress. Note that we have often found a Run Demo to be a suitable opportunity for general "back patting", boosting morale after a difficult job well done.

10.9. Asking for Help

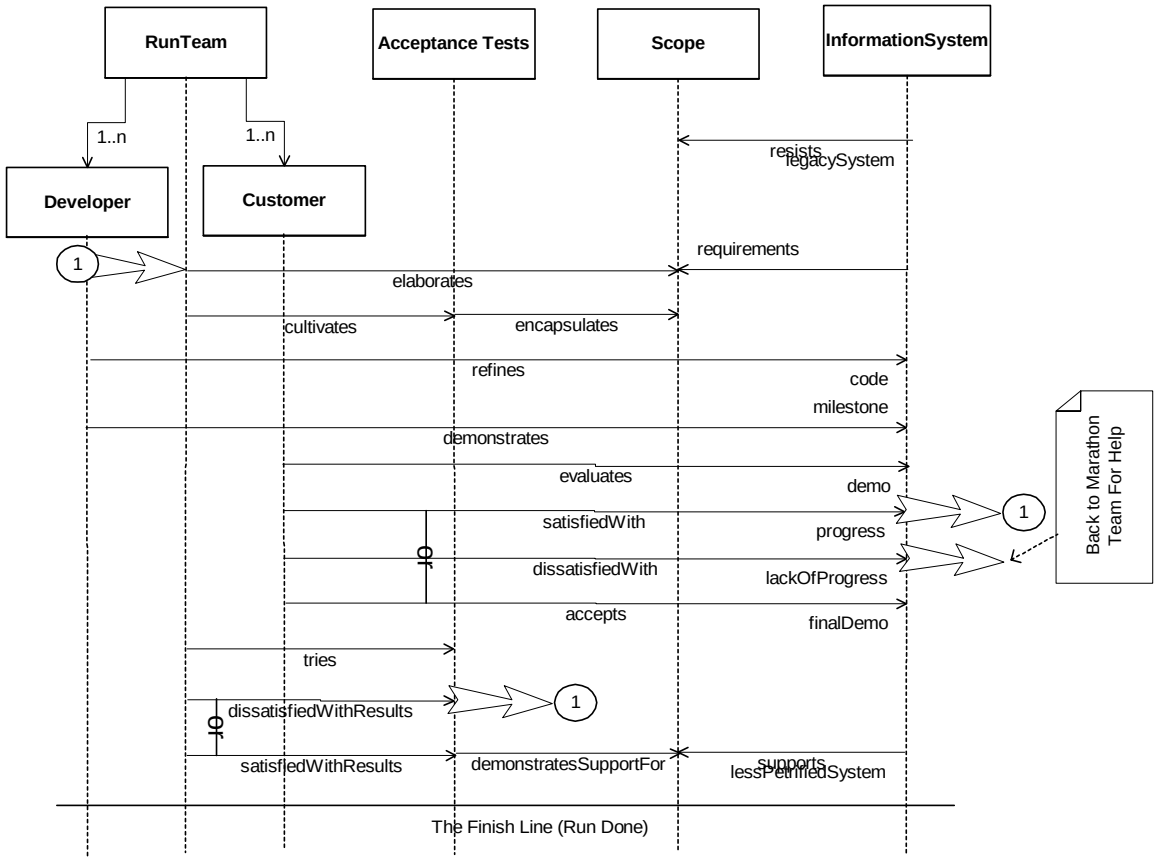

Figure 10.6 – Typical Migration Run

Figure 10.6 summarises a typical Migration Run as described thus far. Again, we must emphasise that we are not dogmatic here; the only mandate is that a Run team successfully injects support for the its scoped aspect of resisted business value in the system. However, in summary, typical Runs in which we have participated involve developers and customers elaborating requirements and writing Acceptance Tests that encompass them. Developers then refine the systems implementation to satisfy those Acceptance Tests (precisely how they do this is the focus of the next chapter). Demonstration of successful Acceptance Tests indicates Run completion, whereas Run team dissatisfaction with their own progress can stimulate appeal back to the Marathon team for help.

Thus, there are times when others, who can see the bigger picture, need to be brought in. In particular, we have already seen that the whole Marathon team meets frequently in short Marathon Steering sessions. It is during such sessions that major concerns emerging from individual Runs are expressed, and their resolution negotiated. A typical Marathon Steering session consists of team members reporting progress on the Runs upon which they are focused, expressing how far they have progressed, how far they are from demonstrating success, and what new problems have emerged that require team-wide negotiation. Particular attention is paid to sharing experience emerging as the understanding of a given Run unfolds.

Note that only pressing Run-level issues are brought to the table in Marathon Steering sessions, to prevent these meetings from being bogged down in detail that is only relevant to individual Run teams.

Our most common need for Marathon-team-wide negotiation has been where somebody is “blocked” within a Run because they need completion of some (possibly new) feature in some other Run before they can continue. Since blocking is a major impediment to progress, negotiation of tactics to unblock a blocked developer is given highest priority during Steering sessions. We have realised, then, that an important responsibility for a Run team is to be aware of impediments to their own productivity. Thus if any impediments cannot be resolved internally in the Run, or by a brief direct collaboration between the inter-Run individuals concerned, they can be raised at the next Marathon Steering session for immediate resolution by the whole Marathon team

The second most common Marathon-team-wide concern has been where a new external requirement requires a re-negotiation of the current areas of concern and their relative priority. In XP and Scrum it is argued that newly emerging functionality should not be

allowed to interrupt a team in progress and should, instead, be backlogged until current work in progress is complete. In practice, though, we have found that in some cases it has been necessary to suspend, scrap, or completely rework some Runs where a new requirement "really put a spanner in the works" and required a rethink. For example, there was an early commitment by EDP management to Internet Explorer as a new front-end to legacy applications. The Run team focusing on the user interface then developed libraries which exploited the features of that particular browser to the full, and other members of the Marathon team - focused upon other Runs - interacted with those libraries. Mid-way through the Marathon, an influential customer injected a new requirement for Netscape Navigator support. This was a disaster as far as the user interface Run team was concerned, and required a complete rethink of the libraries they had developed and the interface to them. This was raised at the next Marathon Steering session, and it was agreed that the user interface team would immediately re-focus on a browser-portability Run. The highest priority of developers addressing all other affected Runs was then to adapt their designs to absorb the (ultimately substantial) library interface changes.

The third most common team-wide negotiation in Marathon Steering sessions that we have seen is where a Run team has uncovered either a new area of concern, requiring a new Run, or a substantial change to the scope of an existing Run. For example, part of our migration work involved breaking a particular legacy system up into high-level components that could be "talked to" via abstract remote connection proxies. It emerged from one Run that the need to create and destroy proxies was frequent, and the resulting start-up and shutdown time of back-end components was prohibitively expensive. This became a real showstopper, and was raised at a Marathon Steering Scrum as a major risk. As a result, a new area of concern - High Performance Connectivity - was created and given highest priority. Two senior developers made immediate commitment to that concern and (after much experimentation over a two month Run) developed a transparent cached connection mechanism that amortised the start-up and shutdown time across multiple proxy requests and thus increased performance ten-fold. A second example of a showstopper is the credit card payment requirement, mentioned above, which became so overwhelming that there was little choice other than to delay it for a future Marathon. This allowed the current Run team to refocus their effort on a Run of reduced scope that was ultimately achievable.

10.10. Sketching Requirements

Returning to the nitty-gritty of a Run in action, we noted above that, throughout a Run, developers negotiate with customers to flesh out the detailed requirement for the Run. During this dialogue we have occasionally found it useful to partially sketch out, as domain explorative models, the business processes that include the resisted business process change for which this Run is responsible. For example, to support Back Ordering, it may be useful to sketch out at a high level the business process for which Back Ordering needs to become a part.

To enable sketching of business processes we have derived a novel notation[46], with a timethread inspired [Burr and Casselman, 1994],[Burr and Casselman, 1996] enhancement to statecharts [Harel and Politi, 1998] oriented more towards business process understanding than to an implementation-oriented viewpoint [Lauder and Kent, 2000d; Lauder and Kent, 2001b].

Standard statecharts are particularly suitable for capturing implementation-oriented system dynamics. Unfortunately, when using statecharts to capture requirements, expressive weakness in the notation (see below) forces us to add features to models that are not intrinsic to the business domain being modelled.

As an example, taken from [Lauder and Kent, 2001b] and originating from our own migration work, when an order is being processed, it passes through various notable states in that processing. An expert in order processing would, for example, recognize the following states (among others) as particularly noteworthy: Entered (into the system), Picked (from inventory), Shipped (to the customer), Billed (for payment), Settled (by the customer), and Closed. In addition, the order may be: Cancelled, Returned, Restocked (back to inventory), and Refunded.

Of course, these states cannot occur arbitrarily. The statechart notation (see Figure 10.7) enables us to express notable states (here, rounded rectangles), transitions between them (arcs, giving ordering), and the events (arc labels) that cause such transitions to occur.

[46] Note that ee have found tool support for our visual modelling work to be somewhat lacking (see below, and, more generally, chapter 14). This has necessitated recourse to pen and paper, which interestingly has had the (now obvious, but initially unforeseen) benefit that we are no longer constrained to any particular notation that our modelling tool may support. Consequently, we have been able to experiment with notations that might be more appealing to the customer. A further advantage, pointed out to us by Stuart Kent, is that there has been no necessity to define these "customer friendly" notations formally, though this would of course change if we wished to provide tool support for them to allow, for example, model animation, or automated traceability between models.

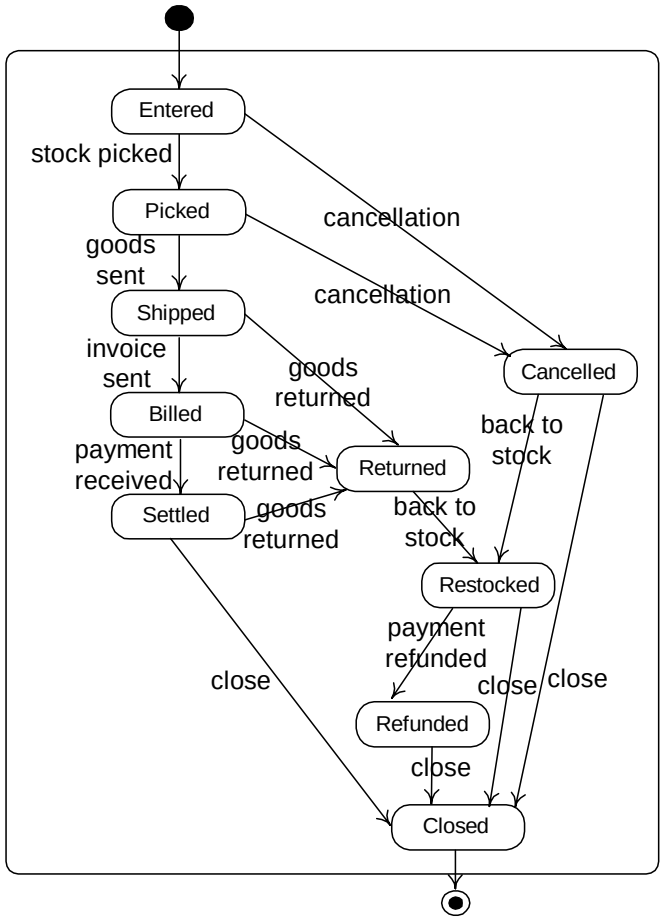

Figure 10.7 – Statechart For Order Processing Dynamics

Unfortunately, Figure 10.7 permits many transitions that a human expert would recognise as inappropriate. We see, for example, that an order may be returned and then restocked anytime after it has been shipped but not yet closed (which is fine). However, according to this statechart, payment for a restocked order may also be refunded even if payment has not actually been received for the order.

To alleviate such problems, we are forced to adorn our statechart with additional artificial states, such as those depicted in Figure 10.8.

Now, Figure 10.8 is a more accurate expression of the order processing business process than Figure 10.7, since it does not permit inappropriate transitions. However, by achieving accuracy we have lost simplicity. Particularly unsettling is that the newly added artificial states are a symptom of the nature of the notation rather than intrinsic to the business process being modelled. For example, a business person would readily use terms like "the order has been Restocked," but is highly unlikely to use phrases like "the order was Restocked after [it was] Paid [for]". Thus, artificial states - although technically correct - both clutter the model space and are jarring aesthetically and intuitively.

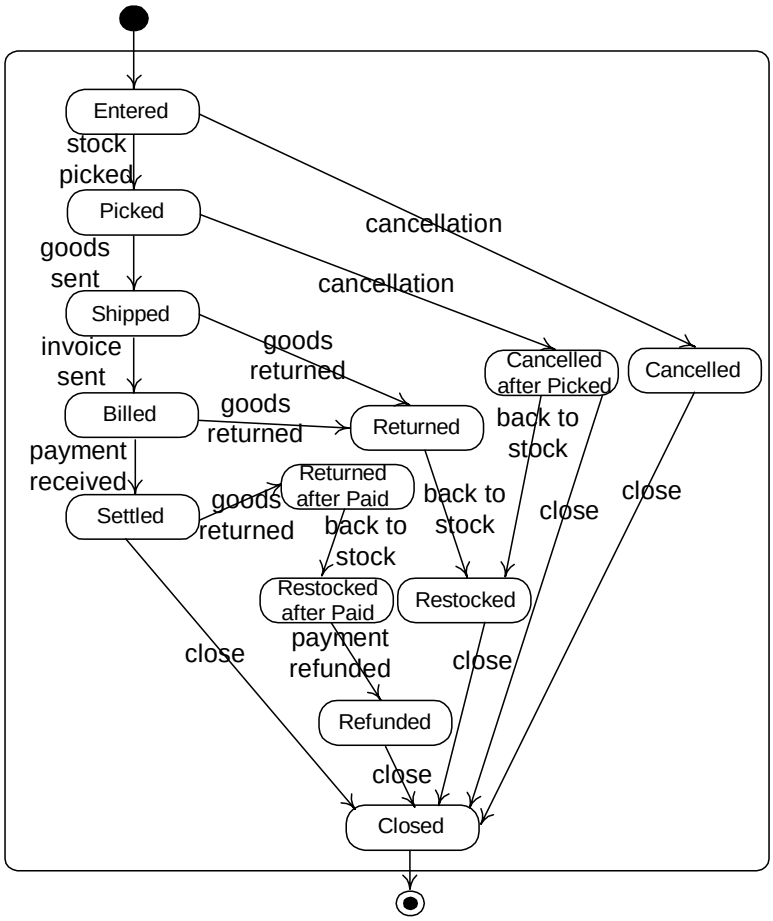

Figure 10.8 – Statechart Adorned With Artificial States

What we really need is a way to capture only those states that are present in the normal vocabulary of the domain (omitting artificial states enforced by the notation), whilst also achieving an accurate model of the business process that does not permit invalid transitions between states.

Our insight (see [Lauder and Kent, 2001b]) was that the problem is not with states, but rather with transitions. Transitions only indicate progress between two states at a time, leading to a very "bitty" expression of a business process. A business process, though, is more than the sum of its individual transitions; there are important constraints on state ordering that span more than two states at a time.

Now, there already exists a notation which captures whole process flows directly - the timethread notation of Burr and Casselman [Burr and Casselman, 1994; Burr and Casselman, 1996; Burr and Casselman, 1996]. Unfortunately, timethreads essentially ignore state. We realised that by blending timethreads with statecharts, we achieve an elegant combination giving state and behavioural flow equal prominence. In Figure 10.9, then, there are no artificially introduced states, as there were in Figure 10.8, since a timethread shows all permitted flows through the statechart.

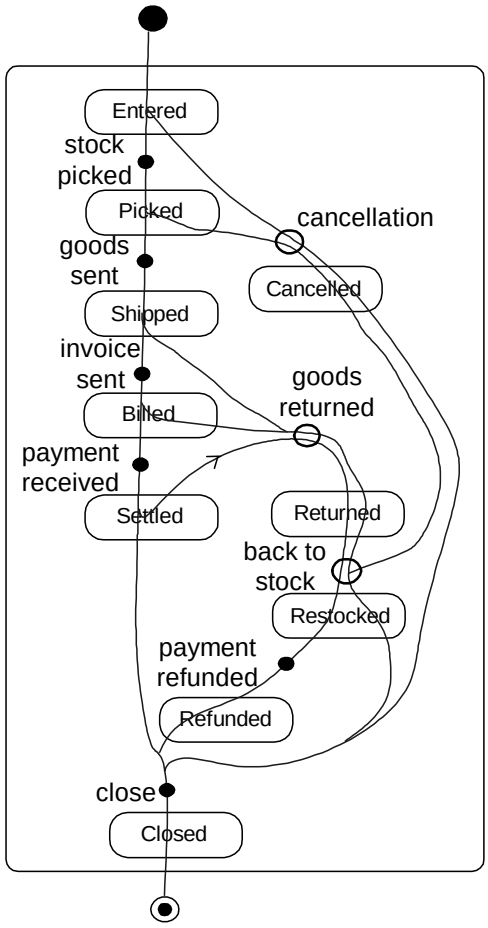

Figure 10.9 – Timethreaded Statechart

Each split in a timethread indicates alternative behavioural flows through the statechart. Which path is taken at such a split is determined by whichever path's triggering event occurs first. Thus, for example, in Figure 10.9, when a product is in the Picked state, it may either become Shipped or Cancelled, dependent upon whether the “goods sent” or the “cancellation” event occurs first. We see in Figure 10.9 that event “wait points” are depicted with small (usually filled) circles. These allow us to distinguish labels that represent event waiting from other forms of label. More importantly, wait points allows several paths to pass through the same event, and by “peeping” into an (unfilled) event circle we can see whether those paths converge (or a single path diverges) at the wait point, or whether (and how) each path progresses through the wait point independently.

Note that we have needed to adorn timethreads with additional capabilities, not shown here, for their blend with statecharts to be effective. In particular, we often need to express “action points” on timethreads, indicating points where processing occurs. We achieve this by adorning timethreads with labelled small filled squares (“black boxes”, borrowed from [Ould, 1995]) akin to our wait point circles. Again, as with wait points, multiple paths may pass through a single action point, and we can peep into an action point to see whether or not paths diverge, converge, or remain separate within them.

10.11. Identifying Context

Figure 10.9 is typical of the type of sketch we may draw when starting, for example, the Kitchenware Back Ordering Run. Now, it is immediately obvious that the sketch does not mention back ordering at all. This is deliberate; we cannot sketch out how back ordering works at the business process level, since the fact that back ordering came from the backlog implies that business process support for back ordering is currently resisted (i.e. stifled). It is only when we have introduced support back ordering that it can actually be part of the whole ordering business process. Our focus at this stage, then, is not premature business process engineering, but rather to map out the current business process foundation upon which we are building. Thus, all we get from timethreaded statecharts is a general idea of the whole business process flows in which our Runs must participate. At this stage, our focus is very much upon establishing a shared context for the Run team, providing a common basis around which ongoing conversations about and, hence, elaborations of the scope of the Run can be based. It is only as a Run unfolds that we can begin to fully comprehend how the intricacies of the resisted requirement we wish to support will affect the nature of the business process of which it will form a part.

10.12. Requirements Negotiation

Once we have a general context for a Run, then, we can begin to negotiate an elaboration of its scope. Our timetheaded statechart sketch is a useful starting point for such elaboration. For example, in Figure 10.10 we highlight the general area in the ordering business process where we think Kitchenware Back Ordering will fit in. We don’t worry too much about the detail at this stage, but it is clear that back ordering must (obviously) precede stock picking. One thing that comes out of Figure 10.10, and that may not be evident if we didn’t focus on whole process flows, is that a product which has not been Picked from stock may be Cancelled (since there is a cancellation timethread prior to stock picking). Since back ordering necessarily precedes stock picking, we see that this constraint probably applies to Kitchenware Back Ordering. Furthermore, we see that when an order is Cancelled before stock picking, that order is immediately closed, without passing through any other states (e.g. Returned, Restocked, Refunded).

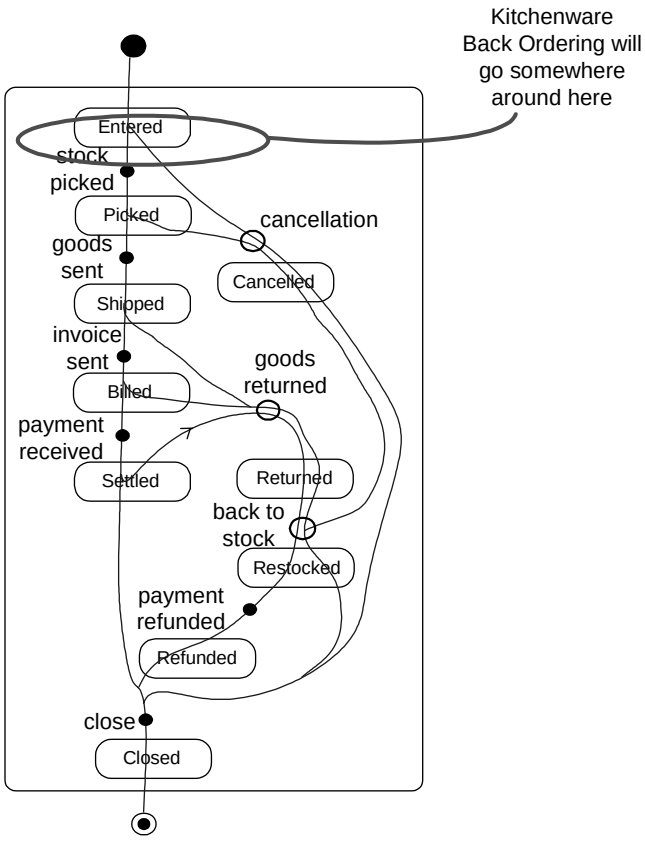

Figure 10.10 – Timethreaded Statechart

Requirements elaboration for a particular Run, then, involves fleshing out the requirements within the context in which it will fit, and in particular extracting the constraints which it must respect.

10.13. Declarative Constraints

When working with customers we have found it sometimes quite difficult to focus attention on concise expressions of requirements constraints. To bring focus and clarity, we have generally found it useful during negotiations to express Run constraints in terms of declarative states that must hold before, during, and after the missing requirement is executed. In other words, developers help customers tease out pre-conditions and post-conditions, and possibly invariant constraints. Since these need to be expressed in a language common to the customer and developers, we have found that it is usually most appropriate to use English narrative, rather than some customer-alienating more formal notation (like Z [Abrial et al. 1980], or OCL [Warmer and Kleppe, 1998]).

10.14. Appealing to Pictures

Having said that, we have sometimes found English to be rather cumbersome (if not ambiguous). Adhering to the adage that a picture is worth a thousand words, EDP

developers have in the past attempted to support the most cumbersome English test expressions with diagrammatic models using the UML [Rumbaugh et al. 1999]. Unfortunately, this had two negative consequences: firstly we found that customers were generally uncomfortable with the UML notation, and its bias towards object-oriented concepts with which they were unfamiliar. Secondly, the resulting diagrams were often so verbose that comprehending and then managing them became unbearably tedious. Subsequently, EDP developers abandoned diagrammatic modelling of constraints as unworkable.

10.15. Constraint Diagrams

In a Migration Run, then, most requirements are still expressed in terms of rough sketches and English narrative. Particularly troublesome constraints, however, have sometimes benefited from expression using the novel, precise, and visual *Constraint Diagram* notation [Kent, 1997], which expresses declarative constraints succinctly via an extension of Venn diagram notation, including arc-oriented navigation expressions to nominate sets, sub-sets, and set members.

As an example, we witnessed developers working on a Goods Returned Run producing page after page of verbose UML to show how a discrepancy list is formed (depicting goods expected in the warehouse that do not match with goods actually received). Verifying these tediously low-level models with the customer was an exercise in patience. Recasting the requirement as a Constraint Diagram reduced its expression to less than a quarter of a page (see Figure 10.11). Furthermore, the customer told us "That's exactly how I see it in my head."

In Figure 10.11 the circles depict two sets, goods expected and goods received, which are nominated by labelled arcs. Anything that is not in their intersection forms part of the discrepancy list.

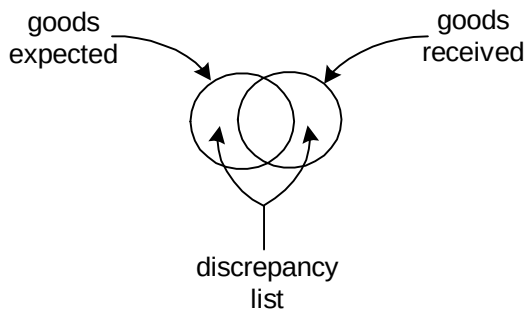

Figure 10.11 – Discrepancy List Constraints

10.16. Contract Boxes

We have also found the *Contract Box* [Kent and Gil, 1998] extension to constraint diagrams particularly beneficial for the expression of more sophisticated constraints. Here, pre-conditions and post-conditions, each expressed as a constraint diagram, form the top and bottom of a box, which encapsulates the behaviour under test. Occasionally, we may "peep into the box" and specify one or more *Sequence Diagrams* [Rumbaugh et al. 1999] to connect contractual pre- and post-conditions [Lauder and Kent, 1998].

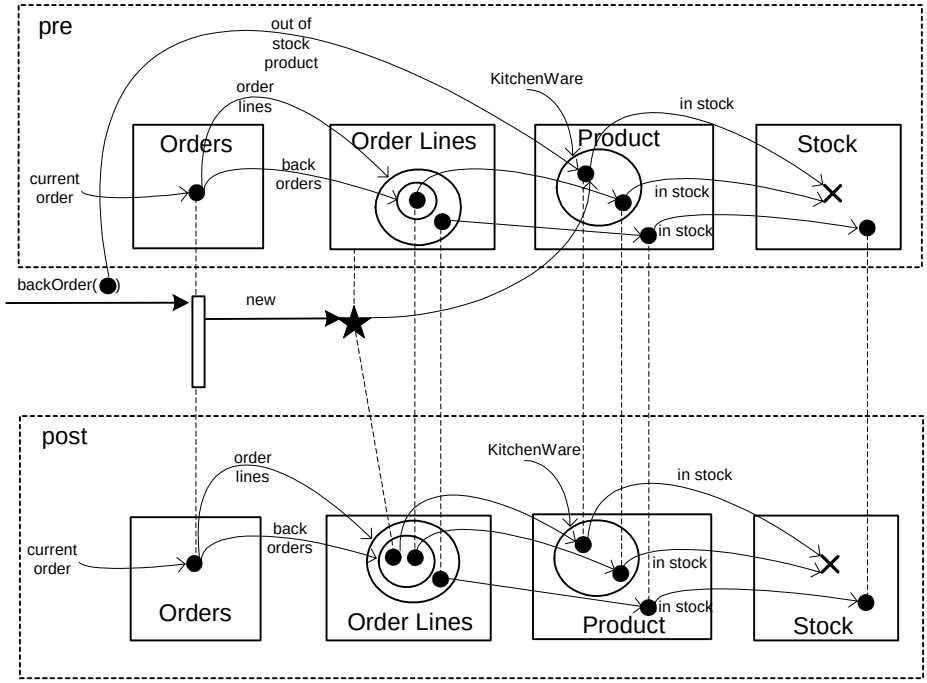

Figure 10.12 – Contract Box Depicting Kitchenware Back Ordering

As an example, Figure 10.12 expresses a (somewhat simplified) Contract Box for the overall Kitchenware Back Ordering operation, which forms the high-level scope of one of the Runs from Figure 10.3. In Figure 10.12, circles represent sets, dots within circles represent set members, and arcs depict association. A series of connected arcs depicts a navigation path. The top-most dotted rectangle encapsulates the pre-condition for the operation and the bottom-most dotted rectangle encapsulates its post-condition. The sequence diagram connecting the pre- and post-conditions forms an abstract specification for the interactions that transform the pre-condition to the post-condition. We see, then, that the current order has a set of order lines, each of which is associated with a product. A subset of those order lines contains back orders for products, which are currently out of stock. A stated pre-condition for back ordering a product is that the required product is both an item of kitchenware and is also out of stock. As a consequence of back ordering, the requested out-of-stock kitchenware product is

referred to by a new order line (depicted by a star) in the back orders of the current order.

Note that each contract box only captures a partial model of the overall constraints that must hold. To capture other constraints, such as the invariant that each order line must include a required quantity for the product on order, we will need additional contract boxes and constraint diagrams. Furthermore, as we elaborate requirements for a Run, we typically uncover the need for constraints that we had not even contemplated at the beginning of the Run. For example, what happens to a back order when its out-of-stock product arrives in stock? What happens if the arriving stock can only partly fill the back order? Do we ship out all the arriving stock and reduce the back order quantity, or do we wait until the order can be satisfied in full? As we slowly negotiate answers to such questions, we enrich our body of constraints covering the Run.

10.17. Reusing Precise Visual Models

Interestingly, as requirements were elaborated over numerous Runs throughout our migration work, we began to see many opportunities for reuse of given Timethreaded Statecharts, Constraint Diagrams, and (in particular) Contract Boxes, via both renaming and inheritance. As an example, support for goods returned by customers can be specified (in part) in much the same way as new goods receipt. Indeed, we have been able to generalise Contract Boxes (and, in fact, UML [Lauder and Kent, 1999b]) further by enabling a model to access its own meta-model [Lauder and Kent, 1998]. This facilitates high levels of reuse for abstract visual models, and is particularly appealing for reusable patterns of models expressed as constraints [ibid.]. Unfortunately, we have found in practice that, as the number of models increases, to gain maximum benefit from this approach we need substantially better tool support than is currently available. As one developer has said to us, “For a high-tech industry, the tools available to us are surprisingly low-tech”. In the meantime, cumbersome paper based approaches and, indeed, program code, have proven more viable than current tools for constraint capture and other domain-explorative work. We expect that adequate tool support is still many years away, and we see a pressing need for much more work in this area (see chapter 14).

10.18. Acceptance Tests

Throughout requirements negotiation, a body of *Acceptance Tests* for the missing requirements is negotiated with the customer and (ultimately) satisfied by the

developers. Acceptance Tests are simply tests that will only succeed if the missing business value is successfully incorporated into the system.

Acceptance Tests can be thought of as an agreed specification for the resisted functionality that the developer is to inject into the system. In general Acceptance Tests focus upon exercising the constraints negotiated during requirements elaboration. In a sense, then, Acceptance Tests act as a contract between the developers and customers in a Run team, forming an (often executable) expression of their negotiated requirements for the Run.

As an example, at the very least, an Acceptance Test for Back Ordering of Kitchenware, would probably submit an out-of-stock product to the Back Ordering operation (respecting the operation's pre-condition), and check that, as a result of the operation, that product is referenced by a new order line in the current order's back orders (testing the post-condition). Of course, other Acceptance Tests will need to exercise the constraints that hold when a back-ordered item arrives in stock, and so on.

We borrowed the term Acceptance Test from XP [Beck and Fowler, 2001], although the term was in relatively wide use before XP. In XP it is implied that customers write Acceptance Tests. We have found, however, this often doesn't work well in practice. In particular, although customers certainly (eventually) understand what they are looking for in the system, they are rarely experts in specifying requirements in the form of executable tests. The quality and coverage of Acceptance Tests improves considerably, in our experience, by having a software professional elicit them in collaboration with customers. The developer and the customer then effectively *test the tests* by walking through them (executing them manually), brainstorming for holes within them, and filling those holes with better tests. Consequently, we make no attempt to automate translation from requirements to tests since, apart from having no tools available to do this, we see conversations about Acceptance Tests are essentially conversations about system requirements. Holes in one imply holes in the other. Elaboration and "debugging" of Acceptance Tests, then, brings with it an invaluable elaboration and "debugging" of the requirements for the Run. Thus, the Acceptance Tests themselves are, in a sense, our primary recording of the agreements emerging from requirements negotiation. The fact that Acceptance Tests are often executable, then, means that we can effectively run the delivered code against its specification.

Note that XP implies that Acceptance Tests can be specified completely quite early on. Our experience, though, is that this is frequently not possible if the tests are to provide

reasonable coverage. In particular, as noted above, Acceptance Tests essentially encapsulate elaborated requirements. Since requirements tend to unfold throughout the development process, so must the tests that reflect them. It is not uncommon, in our experience, for Acceptance Tests to be still under evolution until right before final delivery of a completed Run.

We leave it up to each individual Run team to decide what Acceptance Tests they need, what shape they will take, and how they will implement them. We have seen many Run teams automate the execution of their Acceptance Tests, either by hand-coding them, or with the assistance of testing frameworks such as the Junit testing tool [Junit, 2000] made popular by XP. However, not all Acceptance Tests are easily automated; complex interactions with a dynamic graphical user interface, for example, are sometimes difficult to "code up". We have found, then, that some Acceptance Tests (and, indeed, other forms of test, such as Regression and Progression Tests – see chapter 11) are best specified as evolving interaction scenarios to be pursued manually. Furthermore, we have found that even functionality that could be tested automatically often benefits from supplementary manual testing.

Advocates of fully automated tests would no doubt argue that recourse to manual testing reduces accuracy of repetition, thus undermining test rigour. Our experience, though, is that humans performing manual testing are rarely acting as scripted automatons but rather are acutely sensitive to their environment, and are able to "sniff out" suspicious behaviour or unforeseen avenues, revealing fresh insights with which to bolster our tests and our testing strategy on a continual basis. Furthermore, one of major advantage of manual testing over automated testing is that human testers are alert to "unintentional gaps in the stated requirements" [Bach, 1999b], and can roll out tests on the fly to expose them. James Bach calls this process Exploratory Testing [Bach, 2001] – simultaneous test design and execution – and sees the process as more rigorous than random and ad-hoc testing since it can actually be pursed quite methodically [Bach, 1999a].

10.19. Fraud

Acceptance Tests, once satisfied, are taken by the customer as evidence that the legacy system has now been successfully migrated to one that embraces the previously resisted functionality. Note, however, that at this stage there is no evidence that the missing functionality has not simply been force fitted into the system via a devious hack. That will only become clear as later Runs strive to add further functionality in the same part

of the system and uncover whether or not the system really is more adaptable than it was before. Productive Migration can only encourage good practice, but we have found that it cannot prevent wilful fraud.

10.20. Review

Runs break down the work of Marathons into separate areas of concern, allowing focused effort on each, and maximising parallel migration as far as possible. A typical Run team elaborates requirements and captures them in a body of Acceptance Tests, essentially forming a specification for the Run's deliverable. Various sketches can help visualise the emerging requirements and Acceptance Tests as negotiations proceed. Ongoing demonstration of deliverables and other important artefacts throughout a Run allows early re-steering of the Run if it appears to be meandering in dubious directions.

Although a Run team is free to realise the deliverable of their Run however they see fit, we have often found it useful to break Runs down further, into smaller more manageable increments termed Sprints (and possibly even finer ones termed Bursts). As described in the following chapter, the Sprints and Bursts within a Run focus primarily upon a piecemeal approach to both reducing architectural resistance to, and injecting support for, the specific and historically resisted business-value-laden requirement that the Run team has elaborated.

Chapter 11

Migration Sprints and Bursts

If it were done when 'tis done, then 'twere well
It were done quickly.
- William Shakespeare, *Macbeth* I, 7

11.1. Preview

This chapter describes how breaking Runs down into smaller sub-projects, termed Sprints (each lasting a week or two), can help focus migration work and maintain a high pace of productivity. Furthermore, Sprints allow us to demonstrate mid-Run progress, catch mistakes early, and re-steer migration work accordingly.

We can often benefit by breaking Sprints down even further into daily Bursts, which give us a fresh challenge every day, and focus our attention on tackling migration in very small manageable chunks, which apply productive bottom-up refactoring techniques to progressively loosen system petrification. Throughout this work, we generally appeal to cultivated suites of Regression Tests (which help reduce human petrification, by warning us when we are doing some harm) and Progression Tests (that show us we are reducing system petrification - i.e. that we are doing some good).

11.2. Sprints

Runs, as we saw in the previous chapter, typically last for a few months. We have found that in practice, this can be too long term for the focused high-energy effort that migration tends to require. Consequently, many Runs - at least those embracing agile techniques - are broken down (see Figure 11.1) into a series of unfolding short term (usually sequential, but sometimes parallel) *Migration Sprints*. Each Sprint typically lasts one or two (rarely three) weeks. Some Sprints will involve all members of a Run team, whereas others (usually those that can be done in parallel) allow the team to split into smaller Sprint teams.

As shown in Figure 11.1, then, a long-duration Run is generally partitioned by the Run team into a series of short Sprints, each volunteered for by part of the Run team. Since Runs are generally performed in sequence, each Sprint could in principle involve the whole Run team. In practice, however, we have often found that the number and mix of people in each Sprint team varies, so that some of the Run team is occasionally free to

(where possible) tackle another Sprint in parallel, to help out with other Runs, or to think ahead about future Sprints, or to pursue other work entirely.

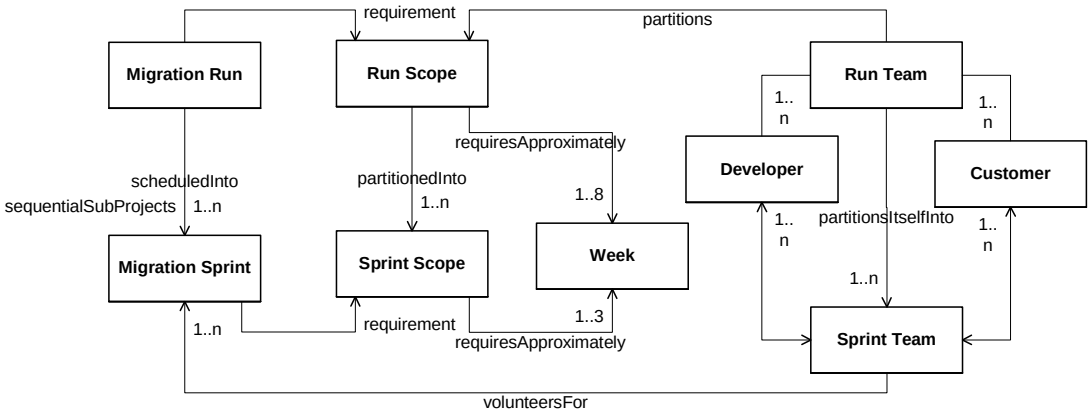

Figure 11.1 – Runs Are Scheduled Into Sprints

11.3. Sprint Scope

In the preceding chapter we noted that the scope for a given Run will typically encompass multiple constraints, elaborated throughout the lifetime of the Run. Early in a Run we will usually have more confidence about the intricacies of some constraints than others. For example, we saw in the previous chapter that for the Kitchenware Back Ordering Run we are confident early on that the back ordering operation involves an out-of-stock Kitchenware product being referenced by a new order line in the back orders for the current order. It makes sense, then, for the constraints about which we are confident to form the scope of the earliest Sprints, leaving other constraints to later Sprints as their clarity improves. For example, although we know we will have to deal with a back order when its out-of-stock product arrives in stock, we can leave decisions about how to handle that until later in the Run. The same goes for handling partly filled back orders, wherein we have to decide whether to perform partial shipment, or wait until the order can be satisfied in full, or allow both options.

Thus, for our first Sprint we can break out from the scope of our Kitchenware Back Ordering Run the constraint originally depicted in Figure 10.11, and reproduced here in Figure 11.2. We have found that breaking Runs into Sprints in this way enables Run team members to maintain pace and crystallise clear short-term goals. This allows them to better estimate migration completion dates, monitor (and demonstrate) progress, catch mistakes early, identify misunderstood requirements, and re-steer the Run accordingly. Without constituent Sprints, we have seen Runs tend to meander. Nonetheless, we have seen some Runs succeed without this approach, but in those cases

the team members involved were generally both exceptionally talented and blessed with relatively stable and well-understood up-front requirements for the Runs they pursued.

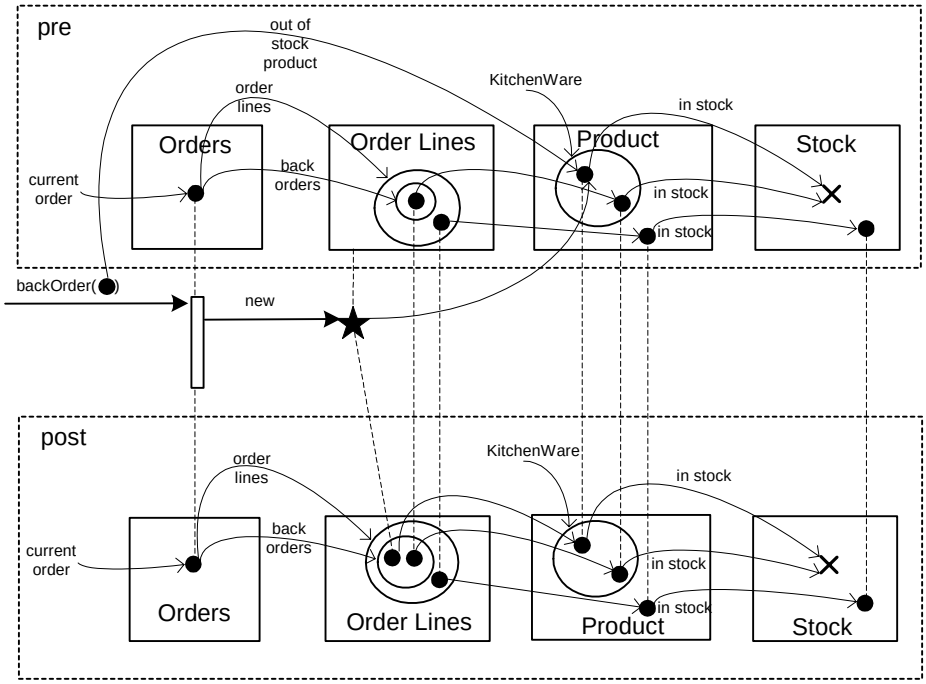

Figure 11.2 – Scope For First Kitchenware Back Ordering Sprint

11.4. Delivering Code

Sprints are primarily about delivering code that reduces architectural petrification with respect to, and then provides support for, the fragment of resisted business value for which the Sprint was scoped[47]. In particular, we find that Sprint teams are usually "developer heavy", with a particular focus on implementation work, since requirements have been mostly elaborated during the Run to which a Sprint belongs. At the Sprint level, then, we have found that our demands on the customer are eased somewhat. This allows them to return to their other responsibilities within the company, whilst remaining available for periodic consultation as needed (to pin-down ambiguous requirements or negotiate slight adjustments in scope), in addition to their involvement in Steering and Demo sessions at higher levels of granularity. XP, in contrast, advocates permanently on-site customers, with full time commitment to the development team. We have found that, in reality, this is rarely achievable, since customers usually have other responsibilities within the company that they cannot afford to shirk.

[47] We leave it up to the Sprint team to decide how to do this, but we examine one effective approach for realising a Sprint deliverable incrementally when we discuss Bursts later in this chapter.

11.5. Sprint Demos

As with Runs, a Sprint team is given effectively complete freedom to cross the finish line in whatever way they deem appropriate. Nonetheless, a Sprint team usually holds frequent (every day or two) informal *Sprint Demos* where they demonstrate to other members of their Run team significant progress in the Sprints upon which they are collaborating. We have found it important to only include other developers in these demos - since only peers can appreciate what is going on "behind" a demo and hence can "marvel" at the creative effort that went in to realising it.

Throughout our migration work, both Sprint and Run Demos have proven to be extremely beneficial to team sociology. Not only do they keep things moving at a rapid pace due to peer pressure and self-pride, but they also help the team to jell by giving them a continual feeling of joint-success and mutual appreciation (a highly effective idea we learned from Peopleware [DeMarco and Lister, 1999]).

11.6. Sprint Steering

Members of a Sprint team work closely together on a daily basis. Their continual collaboration with one another (and with customers on an ad-hoc basis to clarify the intricacies of backlogged requirements), and the relatively short duration of Sprints, means that the need for ongoing "official" Sprint Steering sessions is reduced. More typically, members of a Sprint team can usually sense when there is a need for group discussion, and thus tend to call for such sessions on an ad hoc basis.

We have found that “official” Sprint Steering meetings usually only occur when the Sprint team suspects they cannot be self-sufficient in some particular area. In such cases, there is an accepted tendency to "poach" a member of another Run team (or perhaps a customer) who has specialist knowledge or is in a position to negotiate some compromise. For example, one of our Sprints involved allowing customers to sort tabulated data. This required a short poach of a member of the product-catalogue-management Run to clarify that there would already be a configurable default sort order. Where self-sufficiency and poaching are not enough, the Sprint team promotes an issue to the next Run Steering session (which is never more than one or two days away). Of course, if the Sprint team is completely blocked, we have found that an immediate "emergency" Run Steering session can quickly get things moving again.

11.7. Sprint Done

At the end of a Sprint, a final Sprint Demo demonstrates completion to the whole Run team and possibly (if this was an important milestone) to the whole Migration Marathon team. If this was the final Sprint in a Run, the whole Run team is then ready for another Run (usually in another Marathon), or perhaps some non-migration work. If this was a mid-Run Sprint, the Sprint team is then free to pursue another Sprint within the same Run. Sometimes, though, we have seen a “free” Sprint team blocked temporarily, awaiting completion of a parallel Sprint within the same Run before a new Sprint can be allocated to them. To fill this "idle" time, we have often witnessed members of the Sprint team volunteer their services temporarily to other (parallel) Sprints in the same Run, or even to other Run teams. Interestingly, we have found that if we try to force this, rather than relying on volunteers, motivation tends to drop due to what one developer frequently termed "endless migration slog". Some developers prefer to spend the odd day or two of "idle" time at the end of a Sprint pursuing some "pet" project of their own - perhaps rewriting some code they weren't entirely happy with, or adding some new feature speculatively. We have found that good team sociology should allow such occasional whims.

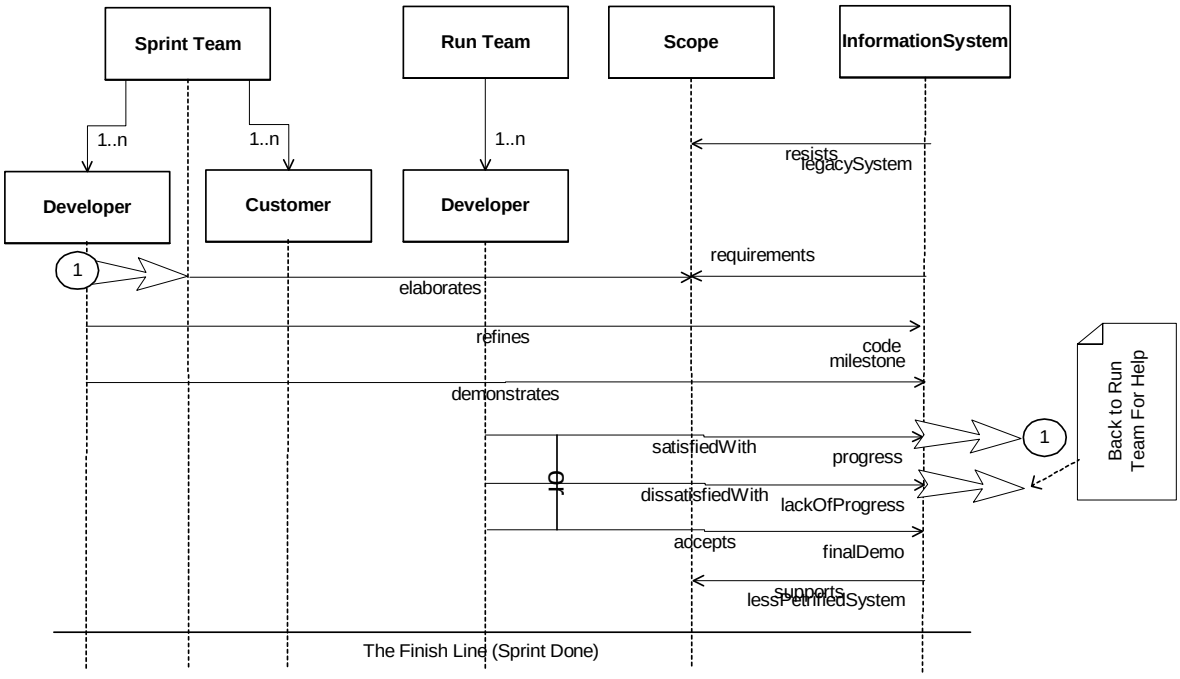

Figure 11.3 – Typical Migration Sprint

In summary, as shown in Figure 11.3, a (developer-heavy) Sprint team elaborates Sprint requirements (where necessary), then the customer fades into the background as developers focus on refining the current implementation to reduce petrification and inject support for the resisted fragment of business value scoped to the Sprint. Other

developers within the same Run team (including those within this Sprint) review progress (primarily via ongoing demos) and, where lack of satisfactory progress cannot be resolved locally, they push emerging concerns back to the Run team as a whole for resolution. When a Sprint crosses the finish line, another Sprint within the Run can begin.

11.8. Bursts

To keep the productivity of Sprints high, Sprint team members often take Sprints lasting a few weeks and split them up even further (see Figure 11.4) into internal *Bursts* (lasting usually one but sometimes two days). We have found that some developers don't feel the need to do this - they like a good long run at something and approach a Sprint intact. Most Sprint teams in which we have participated, though, have tackled petrification through a fast paced explosion of Bursts. As depicted in Figure 11.4, then, a Sprint team voluntarily partitions itself into Burst teams focused upon one (or possibly two) day Bursts. We have found that Bursts can usually scheduled both in parallel and in sequence, according to both developer availability and technical efficacy. Importantly, Bursts are only visible to those pursuing a particular Sprint. We have found that they tend to be too low level and ephemeral to be relevant to others. Note that Burst teams generally only consist of developers, as we will see below, since once we got this close to the code, customers are usually only required for occasional consultation.

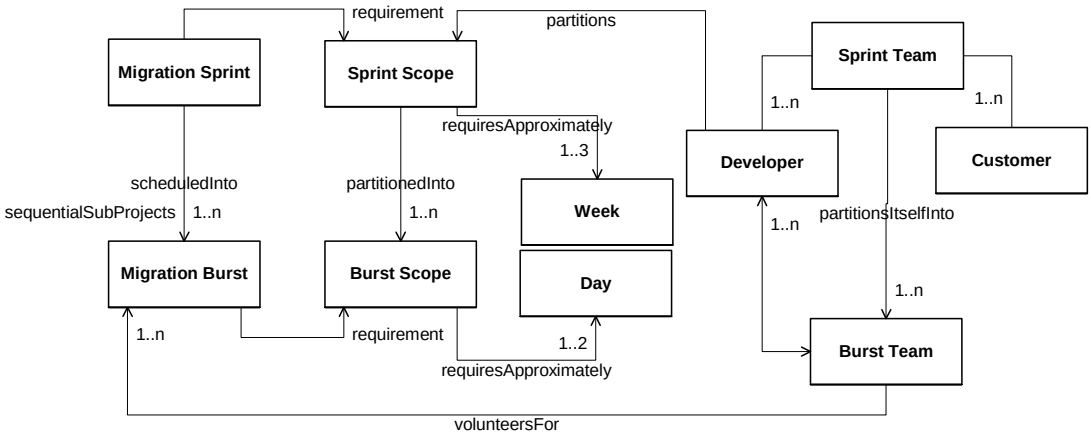

Figure 11.4 – Sprints Are Scheduled Into Bursts

11.9. Burst Scope

There are several compelling reasons for breaking Sprints into Bursts:

- Bursts enable us to separate out concerns, focusing on one thing at a time.

- Bursts allow specialists to be brought in for very short periods, where necessary, with highly focussed attention on their specialist area.
- Bursts enable us to monitor Sprint progress by setting specific milestones and evaluating our success in reaching them.
- Bursts give a sense of progressive completion, which is beneficial to team sociology.

It may not be immediately obvious, given the apparent simplicity of scope of our first Kitchenware Back Ordering Sprint (depicted in Figure 11.2), that there is much potential for, or need to, break that scope into smaller fragments. However, when it comes to implementing that scope, we can at the very least break it down into the three Bursts depicted in Figure 11.5:

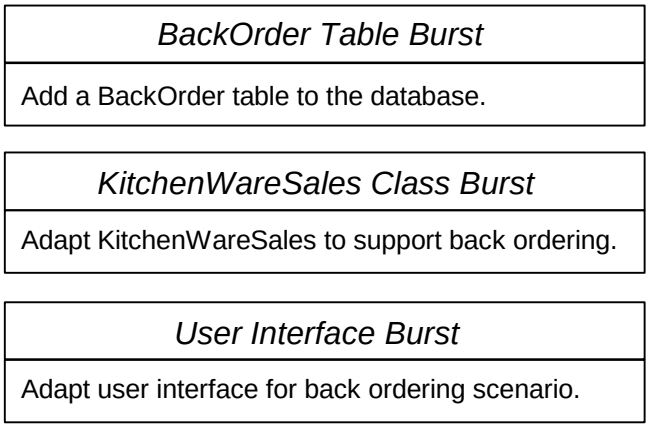

Figure 11.5 – Partitioning Run Scope Into Bursts

For the sake of illustrative example, let us imagine that we have decided to implement the first and second Bursts in parallel. We will leave it to database administrators to tackle the BackOrder Table Burst, and will focus ourselves upon adapting KitchenWareSales to support back ordering. Later in this chapter we will examine this Burst in some detail.

11.10. Migrating Pair

During Bursts, some developers we have worked with have relied on ad-hoc methods, but more frequently Bursts are approached using adaptations of ideas borrowed from XP. As an example, Bursts are often tackled by a *migrating pair*. The essence of pair migration lays not so much in the shared programming effort as in the continual negotiations it encourages between the pair. We have found it extremely effective to pair a developer who knows the current code well with a developer who is new to the code. The first developer is there because there is no substitute for hard won familiarity

with the ins and outs of, and tacit assumptions underlying, the current petrified code. The second developer is there in large part to ensure that the current petrifying practices underlying that code are not perpetuated in the post-migration code. In other words, we have found that an experienced eye and a fresh eye see well together.

We were quite worried about the viability of pair migration at first, but were attracted to it because of the wide acclaim received for its parallel in XP, pair programming. We were certain that pairing could only slow migration down, which a team faced with a growing backlog of requirements, and striving for high productivity, cannot afford to let happen. However, having discussed the experience of Pair Migration with the Burst teams we have worked with, there was a general feeling that, for the most part, pair migration has actually increased our productivity rather than hampered it[48].

To reflect upon why pair migration is effective: Tackling petrification is extremely hard work, juggling many different issues simultaneously. Sharing this workload in a pair appears to reduce the intellectual and technical burden on the individual to some degree. In addition, (as promised by XP) we have seen the continual code inspection inherent in this approach catch mistakes early and thus increase the likelihood of early migration success. We have also seen that when one developer is starting to flag, the other developer in a migrating pair can inspire and keep the pace going, and possibly “take the driving seat" for a while. We have also found that there seems to be a subtle and tacit peer pressure in a migrating pair that helps prevent the laziness that frequently creeps into isolated migration work. In particular, we have seen for ourselves a diminished tendency to "guess" where petrification lies and how it is resolved rather than investigate thoroughly; a pair seems less ego-driven than an individual and, in our experience, seems more willing to ask the code and the customer for clarification. Finally, developers have told us that pair migration prevents them from feeling isolated, encourages team bonding, and generally boosts morale[49].

[48] This confirms recent findings by Laurie Ann Williams, whose Ph.D. research [Williams, 2000] discovered, through a series of empirical experiments, that pairs of developers do indeed work more productively than two individuals working separately.

[49] In addition to these benefits, we see pair migration as offering a valuable resolution to the debate about whether or not there should be a separate “maintenance” team from the original development team (see, for example, [Pigoski, 1997]). Pair migration allows members of the original development team and members of the “maintenance” team to work together in pairs, bringing the advantages that both perspectives offer, whilst also facilitating the gradual transfer of knowledge necessary to permit a seamless “handing over” of ownership, if required, wherein original developers progressively transition away from the mix.

11.11. Regression Tests

In chapter 4 it was noted that developers are often terribly afraid of changing a petrified system lest they do more harm than good. In large part, this stems from the fact that in many legacy systems there is rarely any feedback to inform that you have done damage until the system has been deployed and the user is screaming that critical business data is no longer computed correctly.

To get over such human petrification, and inject courage into a migration team, we need much earlier feedback to alert us when we are "harming the patient". Consequently, before commencing a Burst, we have found it extremely beneficial to develop (and run) a substantial body of appropriate *Regression Tests*. These are tests that the system currently satisfies; covering functionality that the system already supports. This is functionality that the system must continue to provide no matter what migration work is done.

As with Acceptance Tests (see previous chapter) we don’t legislate the form that Regression Tests take. Personally, we have adopted a mixed approach of testing frameworks (both Junit and its derivatives [Junit, 2000], and home-grown) and also manual testing. The most important point, though, is not the precise form that these tests take (we leave that up to the preferences and emerging needs of individual Burst teams), but rather that the tests reveal whenever or not we have harmed the system through unanticipated side effects from our migration work.

Regression Tests, then, help us check at the end of a Burst that existing functionality continues to be supported correctly. Since Regression Tests show us early on when we are doing harm, we can either rollback to the pre-Burst state of the system and start again, or correct our mistakes with a Compensating Burst.

Importantly, Regression Tests quell our fear of working in the dark (wherein we cannot see the unfortunate side effects of our migration work). When we can tell that we are doing harm, we are less afraid of moving forward. We see time spent cultivating and employing Regression Tests, then, as a worthwhile investment in reducing human petrification.

11.12. Refactoring

During a Burst we basically do two things: reduce system petrification by eliminating the legacy system's resistance to the business functionality we wish it to support, then increase business value by injecting that previously resisted functionality into the

system. To achieve our first aim - reducing petrification - we appeal to *refactoring* [Opdyke, 1992; Fowler, 1999b]. Refactoring is a process of incrementally applying a series of small transformations to code that collectively improve the internal structure of that code whilst preserving its behaviour.

Refactoring is essentially equivalent to program restructuring, as investigated by Griswold [Griswold, 1991], although the emphasis there was very much on the viability of tool support rather than on the actual catalogued transformations[50]. Bennett and colleagues [Bennett, 1998] have also investigated a similar approach (program transformation) but with a more ambitious scope, including translation to a potentially different target language in addition to structural transformation. Refactoring, on the other hand, views language translation and tool support as separate (albeit important) concerns from the behaviour preserving transformations themselves.

Of the published agile methodologies, only XP appeals to refactoring; the intent there is to continually evolve the system's architecture "just-in-time" to track changing requirements. In the context of Productive Migration, on the other hand, refactoring is essentially about first progressively peeling back and absorbing into the system's legitimate architecture the layers of implicit architecture that underlie petrification. To achieve this we apply a progression of small, simple, transformations (termed *refactorings*) that incrementally loosen petrification and *legitimise* the part of the system's architecture that is relevant to the resisted functionality we wish to inject. Throughout such refactoring, we run our Regression Tests frequently to ensure that we are not harming the system.

A Migrating Pair is principally concerned with their own Burst during refactoring. Nevertheless, our experience is that they will usually be in frequent contact (during various Steering sessions and informal contact) with other migrating pairs working on related Bursts (possibly in other Runs within the same Marathon). This increases the likelihood that they are supporting one another's direction of migration. For example, it was noted in chapter 9 that backlogged requirements with similar commonality and variability needs are generally scoped into the same Marathon. Constant communication between developers across Bursts ensures they are benefiting from each other's refactoring progress, rather than pulling the architecture in different directions.

[50] Griswold's main argument was that automated tools are necessary to apply transformations accurately, whereas manual restructuring is more error prone. Interestingly, he subsequently backed away from this stance, observing that "[although] methods have been developed for automating the difficult aspects of restructuring … industrial strength tools are still a ways off. However systematic manual restructuring that emulates these methods are [sic] feasible." [Griswold, 1996]

11.13. Refactoring Browsers

Note that there are already a number of automated tools available to assist refactoring work (the most widely acclaimed being [Roberts, 1999]). Such *Refactoring Browsers* encapsulate knowledge of various refactorings, and are trusted by the developer to help apply those refactorings correctly. Unfortunately, all known refactoring browsers are language-specific, and are of little use to those migrating systems developed in unsupported languages. Furthermore, all known refactoring browsers are hard-coded with specific refactorings and are not helpful when other (non-supported) refactorings are applied.

Hopefully, we will soon see language-independent refactoring browsers, or else wider language support, and also refactoring browsers that are extensible with new refactorings by, for example, declarative specification of pre- and post-conditions to a transformation. For now, though, we must rely on manual refactoring.

11.14. The Thinking Refactorer

In terms of the mechanics of refactoring, Griswold has captured a small number of behaviour-preserving transformations for the Scheme language [Griswold, 1991]. Opdyke has focused upon refactorings for C++ [Opdyke, 1992]. Fowler's book [Fowler, 1999b], on the other hand, is a particularly comprehensive reference for Java refactorings. Many of these documented refactorings are clearly specific to certain types of language. For example, Opdyke’s “Change Superclass”, and Fowler’s "Replace Conditional With Polymorphism" assume language support for inheritance. Many refactorings, though, are relatively language independent. For example Griswold’s “Move Expression”, and “Extract Function”, Opdyke’s “Reorder Function Arguments”, and “Convert Variable Reference to Function Call”, and Fowler’s "Inline Temp", and "Consolidate Duplicate Conditional Fragments", among many others, seem widely applicable.

Even without refactoring catalogues, though, our experience has been that developers working at the highest level of maturity (Adaptation) tend to have an intuitive feel for the refactorings that exist and that make sense. Although refactoring catalogues are certainly valuable for inspiration and for vocabulary, the key, we have found, is not to force developers to repeat a cookbook of immutable refactorings unthinkingly, but rather to encourage them to practice behaviour preserving incremental transformation of petrified architecture to legitimised architecture.

11.15. Sample Refactorings

As an example of the efficacy of refactoring, we shall revisit the petrified architectural fragment constructed through a series of hacks in chapter 4. Figure 4.3 from that chapter, which depicted that petrified fragment diagrammatically, is duplicated here in Figure 11.6. Over the next few pages, we will appeal to a sequence of refactorings that progressively legitimise this architectural fragment, reducing its level of inherent petrification in preparation for the later injection of the functionality for the KitchenWareSales Class Burst whose basic scope is repeated in Figure 11.7.

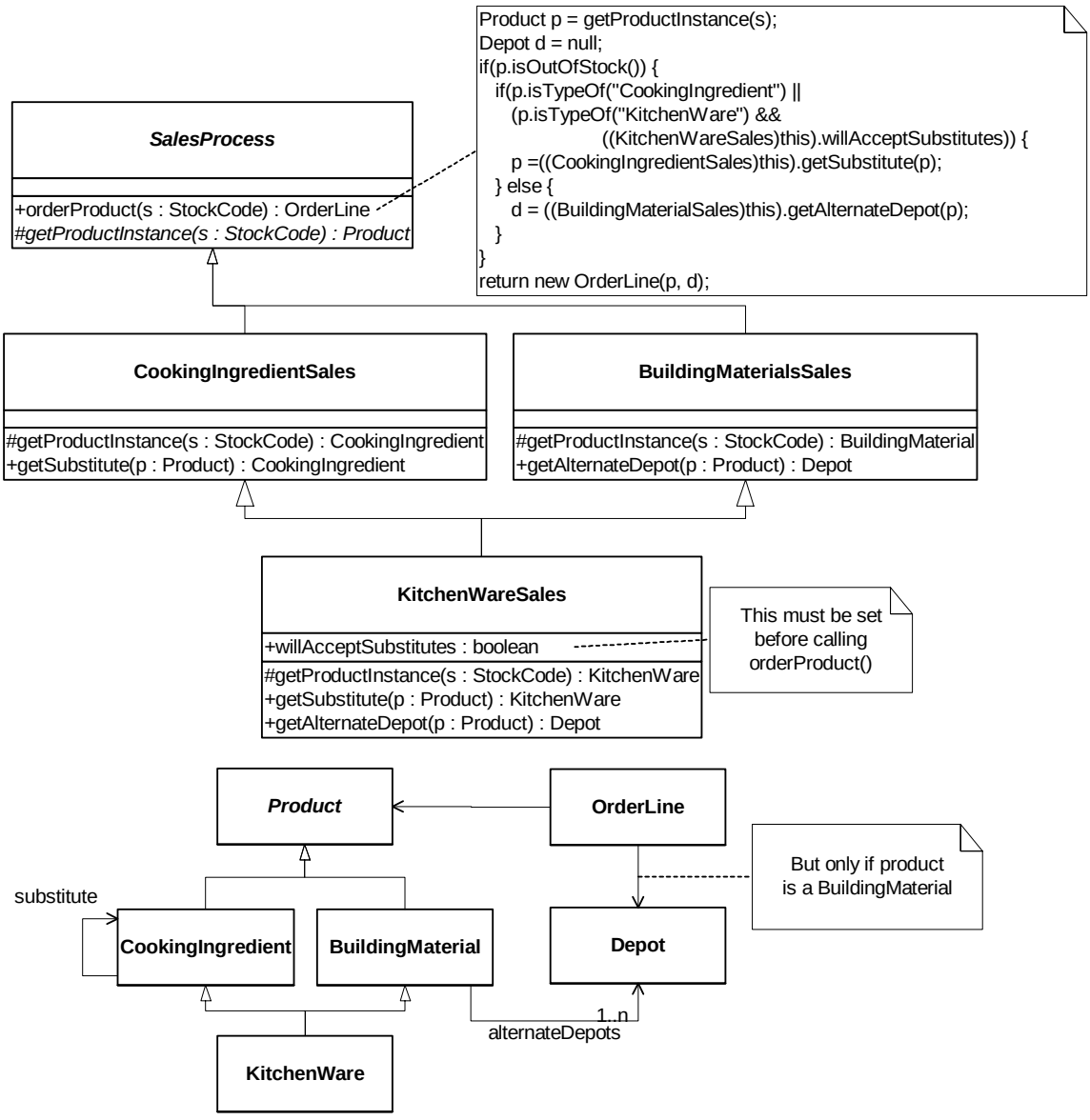

Figure 11.6 – Petrified Architectural Fragment

KitchenWareSales Class Burst
Adapt KitchenWareSales to support back ordering.

Figure 11.7 – KitchenWareSales Class Burst

Upon close inspection of the architectural fragment in Figure 11.6 it is clear that there are several problems that need to be sorted out before we can introduce back ordering into KitchenWareSales. For a start, we need to introduce a BackOrderLine class. Before we can do this we need to sort out the relationship between OrderLine and Product. In particular, we can see that OrderLine has a Depot association for only some objects (i.e. those for which the product is a BuildingMaterial). This is problematic, since there is no protection (other than cautious client programming) against constructing, say, an OrderLine for a CookingIngredient with a Depot for some arbitrary BuildingMaterial tagged on. To resolve this, experience with refactoring immediately suggests Fowler's Extract Subclass transformation, which separates out optional features of a class (such as Depot for an OrderLine) into a subclass. Thus, we create two subclasses of OrderLine, StandardOrderLine and AlternateDepotOrderLine, as depicted in Figure 11.8.

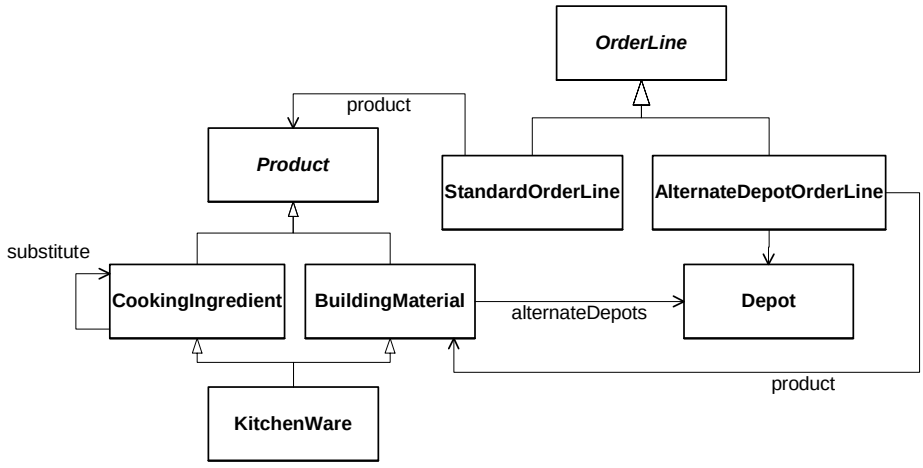

Figure 11.8 – Extract Subclass For OrderLine

Another problem with this architectural fragments is that the inheritance hierarchy for products relies on "likeA" inheritance for KitchenWare in order to allow KitchenWare to be both substituable (like a CookingIngredient) and also obtainable from an alternate depot (like a BuildingMaterial). An unfortunate side effect of this is that KitchenWare will also inherit inappropriate features from its parent classes, such as a sell-by date for CookingIngredient (which is omitted here for brevity). To eliminate these unwanted side affects we can apply Extract Subclass again, this time to the Product hierarchy, thus more correctly capturing KitchenWare using "isA" inheritance, as shown in Figure 11.9.

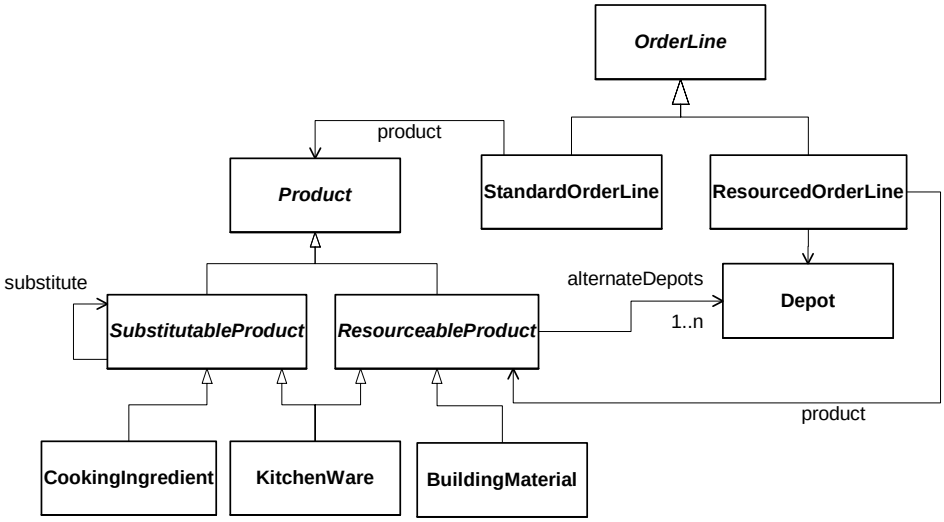

Figure 11.9 – Extract Subclass For Product

Next, we turn our attention to SalesProcess, and recognise that the orderProduct method can be made more readable by separating out a new method (via the Extact Method refactoring) for the code fragment that deals with out-of-stock product OrderLine creation (see Figure 11.10).

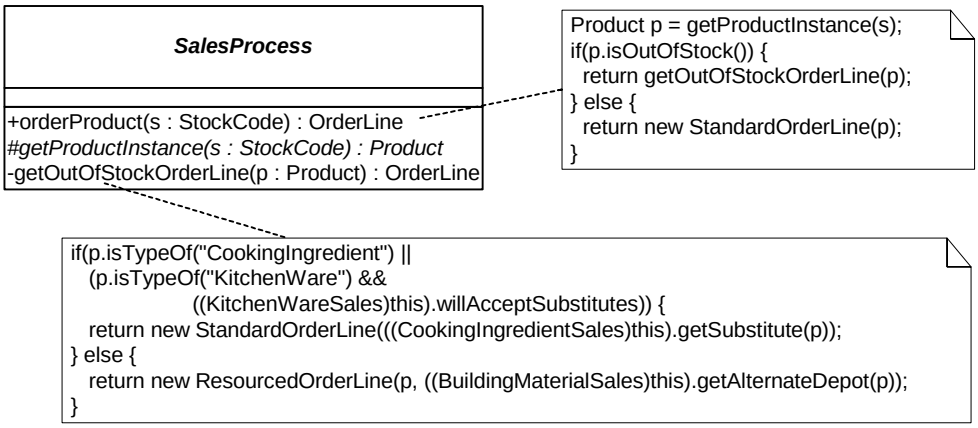

Figure 11.10 – Extract Method getOutOfStockOrderLine

The getOutOfStockOrderLine method depicted in Figure 11.10 is rather intricate, embedding conditional type checking to determine which type of OrderLine to create for an out-of-stock Product. This is problematic, since it means that the method makes assumptions about which subclasses of Product exist, requiring its careful modification whenever a new subclass of Product is added in the future. A second problem is the interrogation of the willAcceptSubstitutes flag of the KitchenWareSales class up in this abstract SalesProcess class, which is clearly a violation of good encapsulation.

To deal with these problems, we initally continued making small scale refactorings to the architectural fragment, for example applying Move Field to move the

willAcceptSubstitutes flag out to the KitchenWare product class, and even a Move Method to move OrderLine creation out to the various Product subclasses. Mid way through this small-scale refactoring., though, came an insight based on a pattern we had seen before. Basically, the insight was that there is an intertwining of out-of-stock product OrderLine creation rules with the hierarchical structure of Product and SalesProcess classes, whereas separating out these concerns would simplify their interconnectedness and enhance their independent maintenance. We were drawn, then, to the Strategy Pattern [Gamma et al. 1995], to separate out from the class hierarchies the out-of-stock product OrderLine creation rules into a family of substitutable algorithms. Thus, we back-peddled our small-scale refactorings after the point reached in Figure 11.10, and introduced a new OutOfStockStrategy hierarchy, as depicted in Figure 11.11 below. The lesson learned here is that the whole process of refactoring is an explorative learning process, wherein our close attention to the architecture, combined with appeal to accumulated experience captured in patterns, can lead to fresh insights with which to improve the architecture at a higher level of granularity. After having such insights, we typically add the deployed pattern to our pattern catalogue (if it was not already contained there), or update our current description of the pattern in the catalogue to reflect our most recent experience[51].

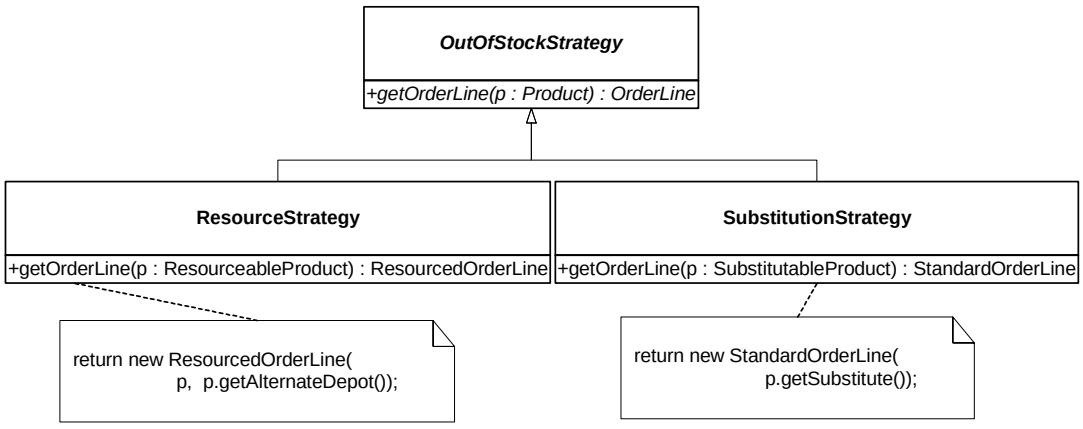

Figure 11.11 – OutOfStockStrategy Hierarchy

At this point, the OutOfStockStrategy includes only two concrete strategies: ResourceStrategy, which only applies to out-of-stock ResourceableProduct instances, returning a new ResourcedOrderLine, and SubstitutionStrategy, which applies to out-of-stock SubstitutableProduct instances, returning a StandardOrderLine for a substitute product. One benefit of introducing the OutOfStockStrategy hierarchy is the potential to

[51] We haven't included Strategy in the pattern catalogues in the appendices, however, since our intent is to include only a representative sampling of the patterns we have come across in our migration work, and besides, Strategy has already been thoroughly described elsewhere (i.e. [Gamma et al. 1995]).

substitute different strategies, where appropriate, when dealing with individual SalesProcesses. Thus, when dealing with CookingIngredientSales, CookingIngredients are substitutable, but not resourceable from an alternate depot, and hence only a SubstitutionStrategy is appropriate. Likewise, for BuildingMaterialsSales, only a ResourceStrategy is appropriate (since no substitutes are allowed). For KitchenWareSales, on the other hand, both ResourceStrategy and SubstitutionStrategy are appropriate, since (as shown in Figure 11.9, above) KitchenWare is both substitutable and resourceable.

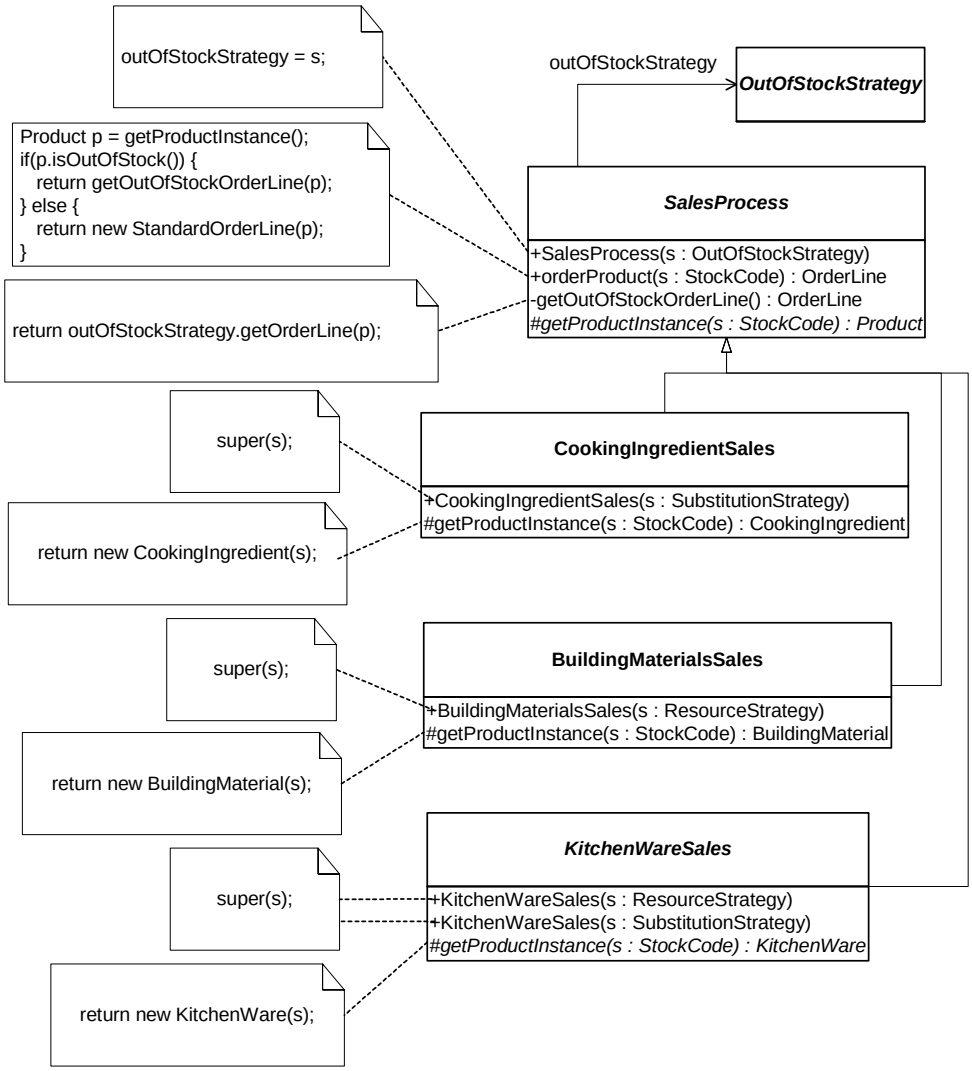

Figure 11.12 – Move Method For Several Methods

Note that the OutOfStockStrategy hierarchy has not yet impacted any of the existing classes, since it has not yet been integrated with them, hence all of our exisiting Regression Tests should still execute, as should new tests we will have introduced specifically for the new OutOfStockStrategy hierarchy.

To take advantage of the OutOfStockStrategy hierarchy, we can (as depicted in Figure 11.12) refactor the SalesProcess hierarchy by performing Add Parameter refactorings to the constructors of the various SalesProcess classes. This enables (potentially substitutable) registration of our (or rather, the user's) currently preferred strategy.

Note that each SalesProcess subclass in Figure 11.12 only permits registration of OutOfStockStrategy subclasses appropriate to it, so that it is impossible, for example, to register a SubstitutionStrategy for BuildingMaterialsSales. Note also that other refactorings have been applied here besides adding a parameter to the constuctors. More specifically: KitchenWareSales no longer needs to be derived from CookingIngredientSales and BuildingMaterialSales (an unsatisfactory form of LikeA inheritance), and instead more properly is directly derived from SalesProcess. In addition, the getOutOfStockOrderLine method of SalesProcess is now, of course, much simplified - removing intricate business rules, and instead delegating responsibility for appropriate OrderLine creation to the currently registered OutOfStockStrategy object.

In a strict interpretation, refactoring should leave the interface to an architectural fragment completely unchanged. Changes should only occur to the internal (hidden) structure. We have deliberately violated this strict interpretation here, since adherence to the current interface forces policy decisions (about the preferred OutOfStockStrategy subclass) to be made internally. Rather than imposing policy on the client, the client is clearly better served by allowing them to make policy decisions themselves. Thus, in addition to improving the internal structure of our architectural fragment, the main benefit to the user of the approach taken above is that client code (rather than SalesProcess classes) is now in control of which OutOfStockStrategy gets applied. The main downside of this approach is, of course, that it requires changes to client code to pass the preferred OutOfStockStrategy to the SalesProcess subclasses. However, by establishing a sold base of Regression Tests for the client code that requires changing, we can highlight where changes to client code have done harm to the system, and make appropriate remedial changes to ensure that Regression Tests are satisfied again.

11.16. Progression Tests

The refactorings we apply during a Burst gradually make the architecture of the relevant part of the system more explicit and more manageable. The *legitimised* architecture reflects more directly the points of commonality and variability inherent in the *currently* supported requirements.

When the relevant part of the legacy system is no longer petrified in terms of its current functionality, we can begin injecting previously resisted functionality (from the backlog) into the refactored code.

Note that refactoring addresses architectural petrification, making such introduction far less arduous than with the session of hacking that would have been necessary had refactoring not taken place. To prevent *human* petrification, however, we need to ensure two things: Firstly, that we can tell that we have injected the new functionality correctly, and secondly, that we haven't damaged anything else in the process. Regression Tests (detailed above) tell us when we have done damage, thus addressing the second concern. To address the first concern, we can appeal to the XP idea of writing a set of tests that fail, but would succeed if the required functionality existed. That is, we write tests for functionality that does not yet exists. In XP, such tests are (controversially) termed Unit Tests, but we have coined the phrase *Progression Tests* to describe them, since when they pass they demonstrate that progress has been made in terms of reflecting requirements that the system previously resisted.

For each Burst, then, after we have legitimised the architecture in preparation for the introduction of missing functionality, we write a set of Progression Tests that cover the scope of that missing functionality. For example, for our Burst of adapting KitchenWareSales to support back ordering, we would write tests that ensured that when a KitchenWare product was out of stock, we could place it on back order, resulting in a new BackOrderLine in the current order. Of course, since there currently is no BackOrderLine class, and no placeOnBackOrder operation, these tests will not yet run successfully. Rather, we take them as our own executable specification of the Burst's detailed requirements, which have yet to be met.

Note that, since Progression Tests (and Regression Tests, for that matter) are internal to Runs (via Bursts), we do not have to "share" them with customers, as we do with Acceptance Tests. Consequently, we can often express them in more concise notations (often simply program code) familiar to a technical audience. Again, we don't legislate here – we leave it up to individual Burst teams to decide upon the specific Progression Tests they require, and what form they take. There are, however, several excellent volumes available, which we have found particularly inspirational in creating effective and comprehensive test suites, with our current favourite being [Binder, 1999].

11.17. Injection

Having developed a set of Progression Tests, we then progressively implement the required backlogged functionality, refactoring (this time in the XP sense of repeatedly introducing a small code change, then micro-rearchitecting to support it cleanly) and re-running Regression Tests as we go. Doing this, we have found that we often uncover "holes" in our Progression Tests, and thus embellish them along the way.

Returning to our example, the refactorings applied thus far have resulted in an architecture that is legitimate with respect to current requirements. We now begin introducing into that architecture support in KitchenWareSales for back ordering, by adding a new abstract Product subclass, termed BackOrderableProduct, and deriving KitchenWare from that class, as shown in Figure 11.13 below:

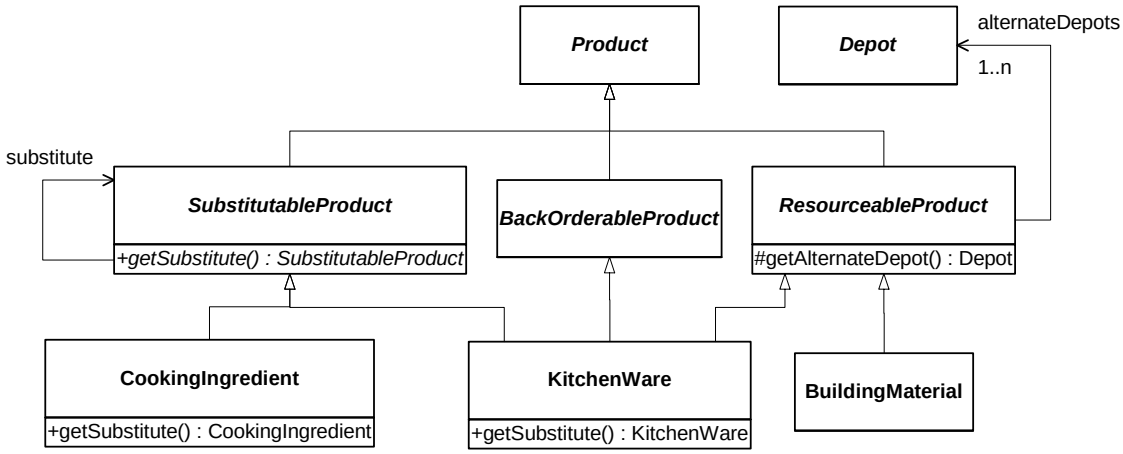

Figure 11.13 – BackOrderableProduct Class

Again, as with all refactorings, we re-run our Regression Tests to ensure that we have done no harm, then we create a new subclass of OrderLine, a BackOrderLine, to refer to BackOrderableProducts, as depicted in Figure 11.14. Since, at the moment, only KitchenWare is a BackOrderableProduct (see Figure 11.13) we are guaranteed that only KitchenWare can become part of a BackOrderLine.

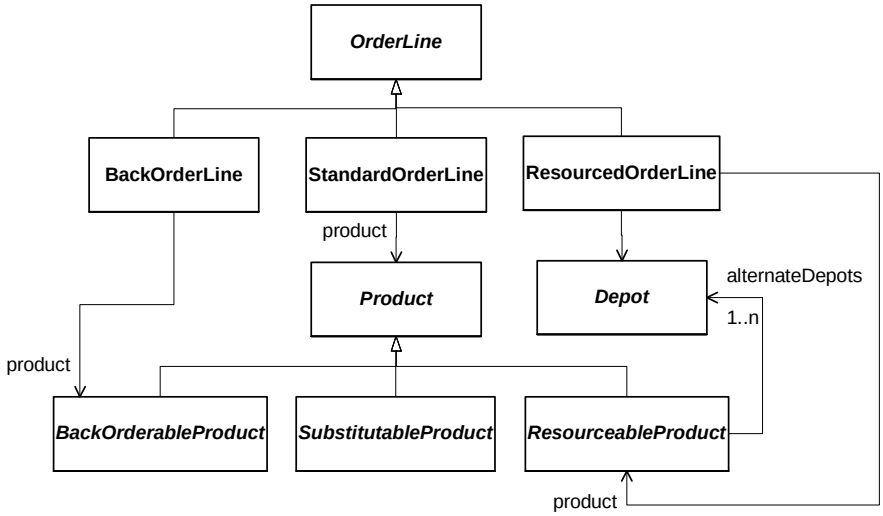

Figure 11.14 – BackOrderLine Class

Now that we have BackOrderableProduct and BackOrderLine classes, and we add support for a new BackOrderStrategy. As shown in Figure 11.15, below, a BackOrderStrategy is a subclass of OutOfStockStrategy that returns a new BackOrderLine.

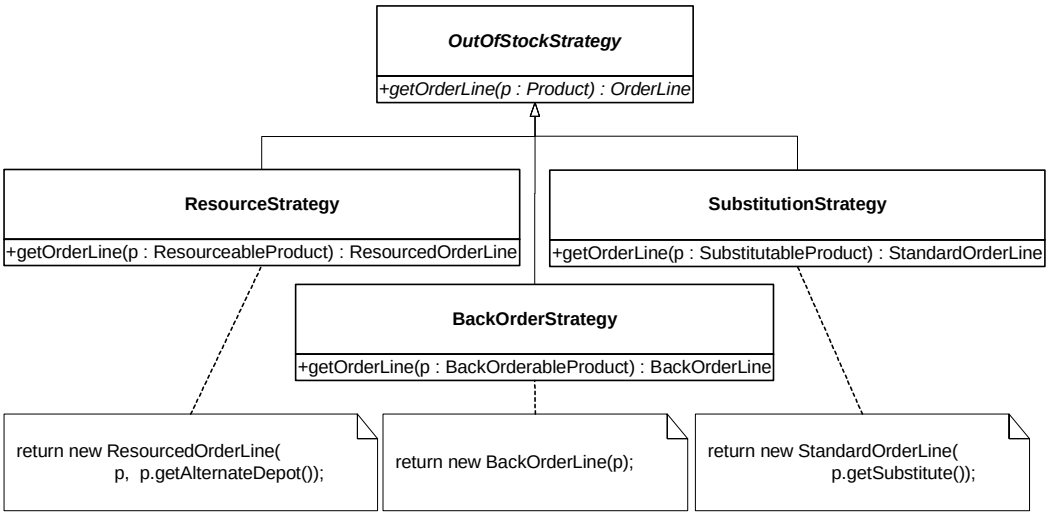

Figure 11.15 – BackOrderStrategy Class

The final change we need to make in order to support back ordering of KitchenWare is simply to add a new constructor to KitchenWareSales, enabling registration of a BackOrderStrategy instance (see Figure 11.16).

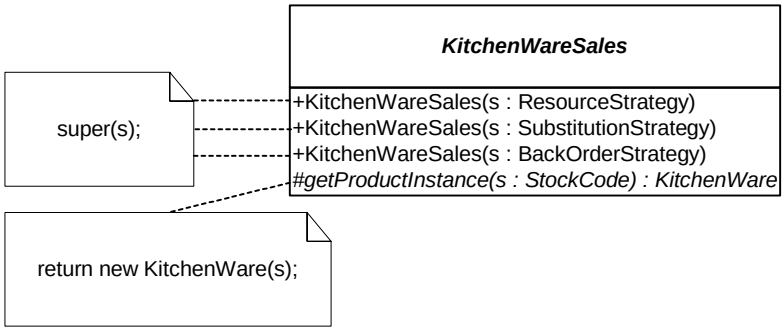

Figure 11.16 – BackOrderStrategy Registration

11.18. Burst Done

When we think we are done with injection, we again run both the complete Regression Test suite across the whole system (to show again that we have done no harm to this or other areas) and the finalised Progression Tests (to show that we have done some good). Our Progression Tests are then merged into our body of Regression Tests to ensure that future Bursts do not upset the newly added functionality. Finally, before moving on to the next Burst, if this was a particularly important or challenging Burst, we demonstrate its completion to the rest of the Sprint team during the next Sprint Demo.

11.19. Showing Off

During our own migration work we have seen Bursts give team members very clear focus each and every day, leading (usually) to a sense of satisfactory completion at the end of each workday and a fresh focus for the next day. Bursts keep the pace of migration success high. Importantly, we rarely allocate less than one day to a Burst - even if the work involved only takes a few hours. This prevents micro-management by peers, and also allows each developer a few hours of free time on occasion to take a breather, pursue some personal interest, and avoid burnout. The fear that this policy will result in a proliferation of simplistic Bursts, where developers only work for a few hours in the morning and goof-off every afternoon has not materialised in practice. If anything, we have seen Bursts increase developer productivity due to the frequent desire among developers to show off to their peers (in dazzling Sprint Demos) just how much they can accomplish in a day or two.

11.20. Mind-Set Bursts

At the end of each Burst, we generally go back to the scope of the Sprint and schedule another Burst from within that scope. Nevertheless, we sometimes find that developers see the results of one Burst as an ideal opportunity to inject some other small fragment

of previously resisted functionality (potentially even a tiny piece of another Marathon) which will be easy to add now that the developers' mind-sets are already focused on the relevant area of code. Thus, for example, at the end of the Burst detailed above, one developer immediately requested an additional (and previously resisted) OutOfStockStrategy. This new OutOfStockStrategy was to be applicable to all SalesProcess subclasses, with no attempt made to find a substitute or alternate depot, but rather an OutOfStockException was to be thrown whenever a product was found to be out of stock. This requirement had not even been scoped into a Marathon yet, although it was agreed that the requirement was real enough. After a short negotiation, we saw that we could realise this change quite easily now (in about an hour), via a quick follow on (mini-)Burst, first adding the new strategy (ExceptionStrategy, which throws an exception) to the OutOfStockStrategy hierarchy, as shown in Figure 11.17. Then, as depicted in Figure 11.18, we add a new constructor with an ExceptionStrategy paramater to each of the SalesProcess subclasses (which, as before, simply pass the parameter on to the constructor of the base SalesProcess class). Finally, we indicate that the orderProduct and getOutOfStockOrderLine methods of SalesProcess may now throw an OutOfStockException (see Figure 11.18 again).

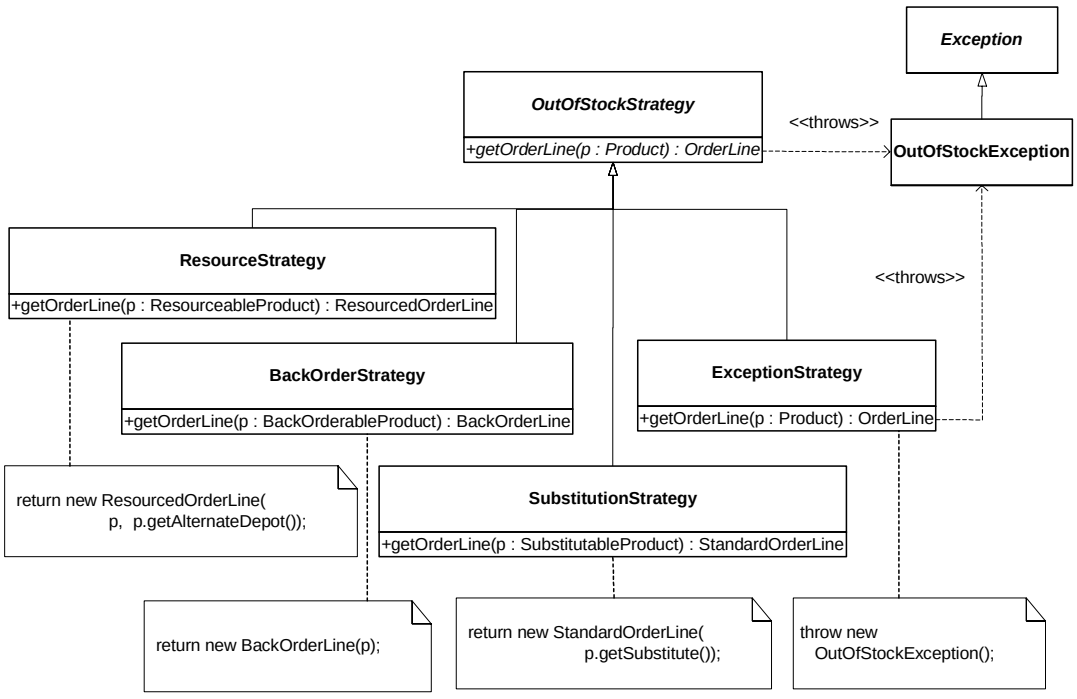

Figure 11.17 – ExceptionStrategy Class

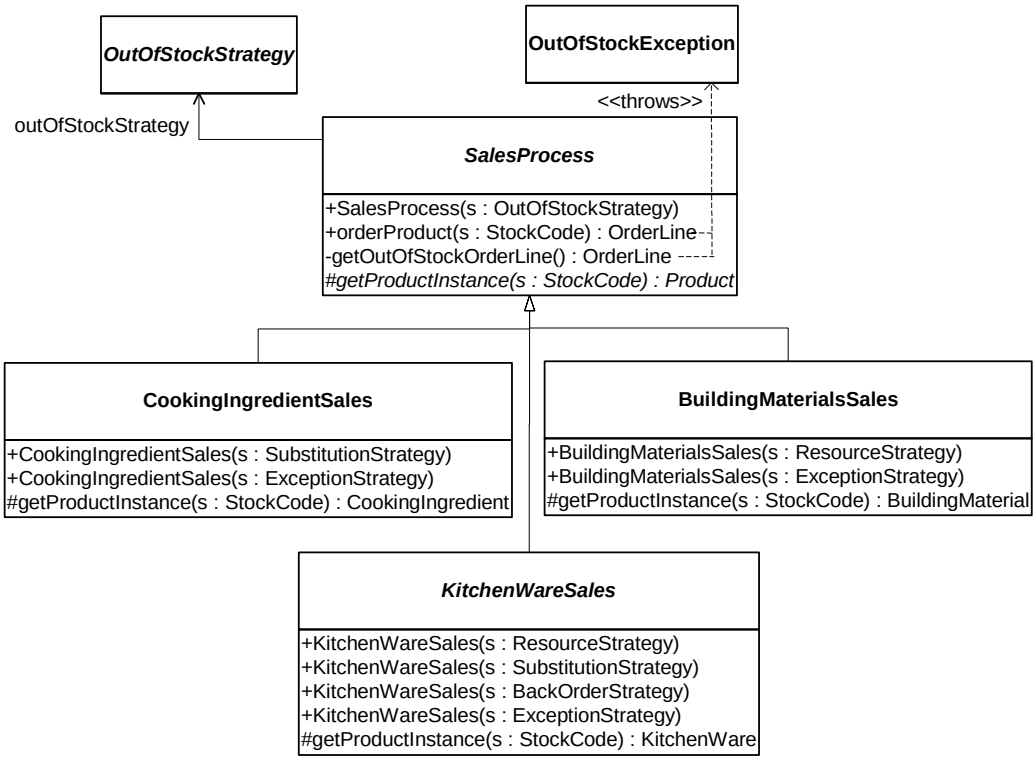

Figure 11.18 – ExceptionStrategy Class

After the short interlude of a Mind-Set Burst (such as injection of ExceptionStrategy support above) we return to standard Sprint steering, inform the team of the completed Burst and the follow-on Mind-Set Burst, and scope the next Burst from the remaining scope of the Sprint. Despite the theoretical potential for an endless series of sneakily inserted Mind-Set Bursts to detract from the progression of a Sprint, we have not found this to be a problem in practice. Rather, we have found that developers tend to act responsibly here, and only agree to Mind-Set Bursts if they really are closely related to their current mind set, and can be implemented extremely quickly without major distraction from other work.

11.21. Review

Tackling petrification by breaking Runs down into Sprints and then Bursts maximises parallelism, keeps us clearly focused every day, and helps us demonstrate progress frequently, catch mistakes early, and correct ourselves when we are straying in a dubious direction. Having help on hand at "higher levels of granularity" in the process (i.e. Runs and Marathons) ensures that we are neither micro-managed nor left wandering in helpless isolation. In terms of the daily practices employed by Burst teams, by continually refactoring to accommodate business value, we can be certain that we are addressing *system* petrification productively. By cultivating and continually applying a rich body of Regression and Progression Tests, we can be confident that we

will ultimately do no harm to the system under migration, thus helping to encourage a culture of fearless migration and alleviating *human* petrification.

As Bursts complete, along with the Sprints, Runs, and Marathons that contain them, the system under migration gradually becomes less and less petrified in terms of the requirements it has thus far resisted. Note that it is probably impossible to say that a system is completely free from petrification - since there is always the potential for further unexpected requirements change to completely invalidate our current commonality and variability assumptions. What we need to ensure, though, is that we do not let the system petrify with respect to its current requirements and that we continually reduce its remaining levels of petrification as new requirements emerge in the future. To keep the system permanently adaptable to (potentially unexpected) future requirements change, and prevent us from clinging to our current architecture when future changes invalidate it, we can appeal to a process of Productive Evolution, described in the next chapter.

Chapter 12

Productive Evolution

Kind is my love today, tomorrow kind,
Still constant in a wondrous excellence.
- William Shakespeare, *Sonnet 105*

12.1. Preview

Earlier chapters tackled the petrification underlying legacy systems that holds businesses back. We have seen that Productive Migration helps us realise adaptable systems, wherein we no longer are saddled with an overwhelming backlog of resisted business value, and can keep pace with newly emerging business process change. Adaptable systems, then, act as change-enablers rather than (legacy system) change-inhibitors. In other words, an organisation armed with newly adaptable systems can begin evolving their business processes effectively, in response to competitive pressures and commercial and technical opportunities, since adaptable systems are able to co-evolve with, rather than resist, such change.

Exploiting this potential for continual business and technical co-evolution[52] can lead, ultimately, to a unified process that has been termed business engineering [Eatock et al. 2000]. This becomes important, since once we have tackled legacy systems in the information systems sense, we can start to address legacy systems in the wider sense that "legacy systems can be said to consist of a business as well as a technical dimension: they contain organisational structure, strategy, process, workflows as well as the software" [Ramage and Munro, 2000b]. Productive Evolution, outlined in this chapter, helps us retain an adaptable system's potential for continual business and technical co-evolution, and thus is an enabler for ongoing business engineering[53].

This is going to be a short chapter, since Productive Evolution does not differ that much from Productive Migration. The main difference is that we no longer have a substantial backlog of requirements that the system has historically resisted. Instead, we are now

[52] Since business activity and supporting technology are closely intertwined, we can never avoid evolution of one impacting the other (see [Mitleton-Kelly and Papaefthimiou, 2000b; Mitleton-Kelly and Papaefthimiou, 2001]. Co-evolution, then, can either be essentially accidental, but unavoidable, or deliberately steered. It is this second sense of co-evolution that we refer to here.

[53] Indeed, [Mitleton-Kelly and Papaefthimiou, 2001] makes a convincing, if rather theoretical, argument for deeply interconnected business and information systems co-evolution being vital to preventing legacy systems from thriving in the future.

concerned with keeping pace with newly emerging requirements change. In short, Productive Evolution, strives to prevent today's adaptable systems from becoming tomorrow's legacy systems.

We are not dogmatic about the practices followed to ensure continued adaptability, since although there was a gap in the literature in terms of responding to petrification (which we believe Productive Migration has filled), there are several existing approaches already available for retaining, or even improving upon, a system's level of adaptability in the light of newly emerging requirements. In principle, then, Productive Evolution could be replaced with any other agile methodology (such as those surveyed in chapter 7), since all agile methodologies share Productive Evolution's aim of remaining responsive to ongoing requirements change. Nevertheless, we have found it particularly advantageous for Productive Evolution to remain close to Productive Migration in terms of the practices it follows. Partly this is because we now have a jelled team intimately familiar with those practices. A more significant reason, though, is that (as detailed below) we frequently follow Productive Migration and Productive Evolution in parallel, with close interaction between them, and we have found that their similarity in principles and practices simplifies their mutual coordination[54]

12.2. Purposefully Non-Speculative

In the last few chapters we have seen that Productive Migration strives to reduce petrification only with respect to backlogged resisted requirements. Thus, even after we have injected adaptability with respect to backlogged requirements, the system will most likely remain petrified against future unexpected requirements changes. This is quite deliberate; addressing petrification speculatively (for imagined future requirements) implies introducing speculative commonality and variability into the system's architecture. The resulting "flexible" architectures may ultimately prove wholly inadequate, as future unanticipated requirements changes invalidate our speculated commonality and variability assumptions. To accommodate future requirements change, then, we cannot rely on speculative architectural flexibility; rather we must wait until real (rather than speculated) needs emerge, then adapt the system's architecture "just-in-time" to reflect the actual commonality and variability assumptions inherent in those real requirements.

[54] Compared to, for example, employing Productive Migration for migration work yet switching to, and striving to coordinate with, a vanilla form of XP or Scrum when addressing new requirements.

12.3. Shifting to Emerging Requirements

After each Migration Marathon has completed, we can do one of two things: either commence another Migration Marathon (if there are any more resisted requirements on the backlog), or take a pause from migration work and address new higher priority functionality. After some reflection it becomes clear that there is actually little difference between the two courses of action. There is no real difference between adding a new feature to the backlog of resisted requirements and assigning it highest priority, and simply calling that feature a new requirement mandating immediate attention and thus pausing migration work. Productive Evolution, then, is essentially a continuation of the practices of Productive Migration, except that the requirements it addresses are newly emerging rather than potentially backlogged for some time.

12.4. Evolution Marathons

During an *Evolution Marathon*, we take a newly emerging requirement and negotiate it with customers. The consensus may be to backlog the requirement. It may, for example be rather low priority, or it may group well with some already backlogged features that we wish to tackle in unison in some later Migration Marathon. If the decision is not to backlog the requirement we scope it out in general terms then assign it a new *Evolution Run*. We may assign other new requirements to other Evolution Runs within the same Marathon. Or, as we have sometimes done ourselves, we may even group some backlogged features into separate Migration Runs, thus forming a mixed Marathon of both Migration and Evolution work (see Figure 12.1).

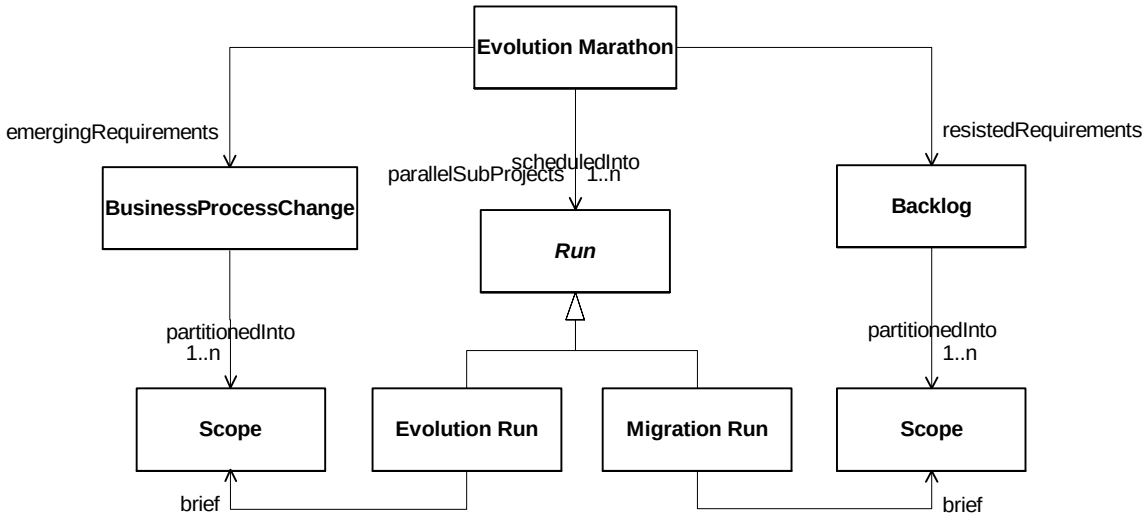

Figure 12.1 – Evolution Marathon Scoping

12.5. Toe-Dipping

As depicted in Figure 12.2, an Evolution Run often begins with a brief *Toe-Dipping Sprint*, wherein we test the water to reveal how difficult it will be to introduce the new feature. Particular emphasis is placed on determining whether the system currently significantly resists that new feature, or can absorb it relatively painlessly. If it turns out that resistance is significant, then we often negotiate in the next Marathon Steering session to halt the Evolution Run and recast it as a Migration Run. One possibility is then to backlog that Run for a future Migration Marathon, and we have certainly done this on occasion. More usually we still proceed with the Run, but realise that (as with standard Migration Runs) we need to first loosen architectural resistance to the feature before it can be injected into the system legitimately.

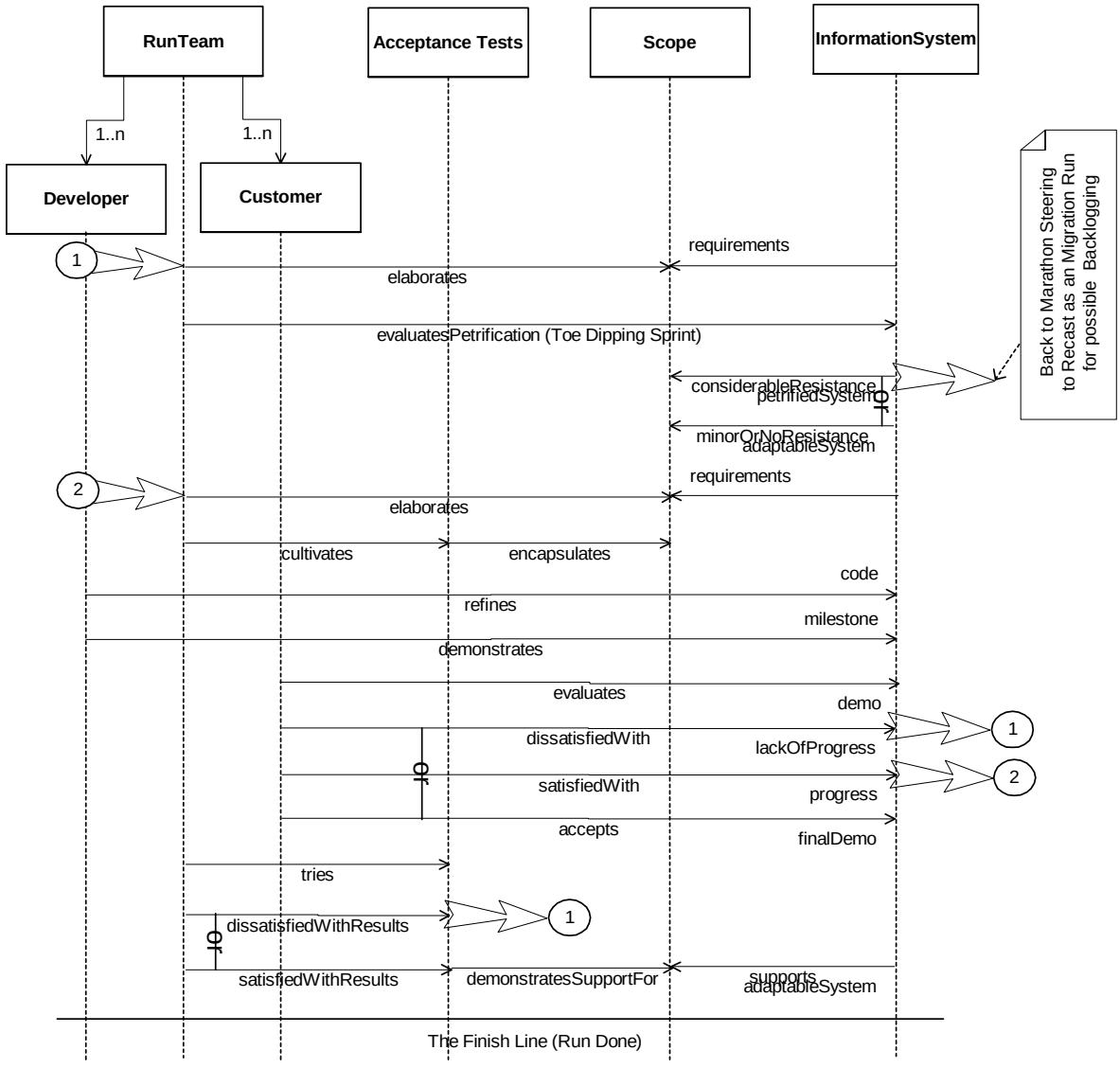

Figure 12.2 – Typical Evolution Run

12.6. Evolution Runs

In Figure 12.2 we see that for a requirement against which the system is not currently petrified, developers work with the customer to flesh out the requirement and negotiate suitable Acceptance Tests. The new requirement is progressively introduced through a sequence of *Evolution Sprints* (and, therein, possible *Evolution Bursts*), which are essentially identical to their equivalents in Productive Migration. Again, as with Productive Migration, Demos and Steering sessions throughout keep things on track, and a final Demo and successful Acceptance Test execution indicate Run Done.

12.7. Small Slips

Interestingly, we have found that Evolution Sprints and Bursts require considerably greater self-discipline than their equivalents during migration. More specifically, we have found a greater tendency to just "hack a feature in" during evolution. Developers seem to be far more willing to force-fit into “almost-right” points of architectural commonality and variability than those that require more considerable effort. We think this is because during migration developers are essentially *forced by the code* to refactor mercilessly, since without considerable refactoring the system will continually resist the introduction of the backlogged feature. With Productive Evolution, however, the system's resistance to the feature is often less great (since where it is great, we usually reassign the feature to a Migration Run). Commonality and variability requirements for the new feature, then, tend not to be greatly misaligned with the commonality and variability assumptions reflected in the system's current architecture.

We have noticed, then, that developers seem far more willing to accept small slips in the architecture rather than large ones. A small hack ("just tweaking around the architecture a bit") is often shrugged off an inconsequential, since "we can always tidy up the code later if we need to". During our own migration work, machismo ("look what I did in just half a day"), and near impossible schedule pressures ("we need it yesterday") have been seen to contribute to the acceptance of small architectural slips.

12.8. Lead to Great Falls

The problem with accepting small slips, as we saw in chapter 4, is that this is precisely where great falls come from. Indeed, tracing back in history, the ongoing acceptance of small slips seems to have been a significant factor in the emergence of legacy systems at EDP. We can see, then, to reiterate an important message from chapter 4, that legacy systems are not built in a day; rather, initially adaptable systems gradually petrify into legacy systems over a relentless series of almost invisible tiny hacks, each of which

could always have been "tidied up later", but none of which ever actually were. Such creeping petrification is often hard to spot, since it tends to start as decay deep inside the internals of an information system. Initially, then, creeping petrification usually has few externally visible symptoms, and hence its damaging effect is often not recognised by anybody (apart, perhaps, from some of those closely involved with the implementation code). Eventually, of course, even externally visible symptoms, such as an unbearable requirements backlog, will be unmistakable. By then, however, it will be too late; we will again have a petrified legacy system, with business needs slipping further and further away from our ability to support them. At this stage, all our hard work will have been lost, and we will again need to begin a progression of Migration Marathons to attempt to bring adaptability back to the system.

12.9. Prevention is Better Than Cure

The most effective way to deal with legacy systems is not to rely on lifeboats when the ship is almost sinking, but to prevent it from sinking in the first place. By accepting no small slips, by being continually vigilant, looking out for signs of creeping petrification, and nipping it in the bud by continually refactoring throughout Productive Evolution, we can ensure that we have precisely the right commonality and variability assumptions in place at all times. Thus, we can be confidant that we are cultivating a permanently legitimate architecture, and that the system will not petrify again in the future.

12.10. Review

The main message from Productive Evolution is that the adaptability of an adaptable system - one that keeps pace with, and thus enables, changes in its requirements - is a function of both that system's architecture and, more importantly, the practices we follow when evolving that architecture. We need to see system adaptability as an ongoing journey, not as a final destination. We are never "done".

Petrification has a habit of creeping back into a system, and we need to be perpetually vigilant against it if we are to prevent stealthy petrification from overtaking us again. Lots of self-discipline, and guidance from the principles and practices of Productive Evolution will help ensure that newly adaptable systems never again degrade into change resistant legacy system liabilities.

Chapter 13

Patterns to the Rescue

The web of our life is of mingled yarn, good and ill together.

- William Shakespeare, *All's Well That Ends Well* IV, 3

13.1. Preview

We have seen in the last few chapters that remaining light on our feet, where developers and customers work closely on a daily basis to elaborate requirements, prioritise lost business value, and deliver just-in-time refactored code incrementally, helps a motivated and jelled team tackle petrification and introduce both resisted and newly emerging business value productively.

We have noticed that the close cooperation and emergent shared understanding underlying Productive Migration and Productive Evolution generally leads to an evolving project vocabulary common across the development team, customers, and other stakeholders. Elements of that evolving vocabulary have deep meaning rich in the semantics of the project. These rich conceptual elements form *patterns*, which encapsulate high levels of potential concept reuse throughout this migration (and evolution) and subsequent ones. It is the formation of, and appeal to such patterns that constitutes team learning, and this is seen as central to a successful migration and evolution effort. The emergence of a body of expertise (captured both in the patterns themselves and, more importantly, in the heads of the project team) contributes to a team that is fitter at the end of the project than at the beginning. In particular, we have learned that assimilated patterns help keep a migration team on the straight and narrow; guiding the team in recognition of, and thus avoidance of, lurking mismatches between the needs of a given legacy system and tempting but ultimately inappropriate courses of treatment.

13.2. Patterns

Much expertise is rarely articulated. Documented *patterns* make potentially reusable tacit expertise explicit and thus available to the less experienced practitioner. Patterns are now widely employed throughout software development. The initial inspiration for this was the work of Christopher Alexander, and colleagues, who sought out and applied architectural patterns in the design and construction of buildings and towns

which were inherently appealing to the human spirit [Alexander et al. 1975; Alexander et al. 1977; Alexander, 1979]. Following this lead, and building upon the Ph.D. work of Erich Gamma (sadly only available in German), the "Gang of Four" popularised the collation and application of design patterns [Gamma et al. 1995], which form commonplace and reusable mid-level architectural abstractions. A great deal has subsequently been written about the application of patterns (of various types) to software development, and many valuable catalogues of patterns have been collated accordingly, e.g. [Gamma et al. 1995; Buschmann et al. 1996; Hay, 1996; Fowler, 1997; Coplien and Schmidt, 1995; Vlissides et al. 1996; Martin et al. 1998; Harrison et al. 2000].

13.3. Reengineering Patterns

Numerous researchers have recently begun to investigate the possibility of applying patterns in the context of legacy system reengineering. Chu, and colleagues [Chu et al. 2000], present an experience report that, not surprisingly, shows how a particular legacy system, saddled with a complicated architecture, improved in terms of maintainability when reengineered to reflect an architecture expressed in terms of well established design patterns.

Linda Seiter, in her Ph.D. thesis [Seiter, 1996], investigates Design Patterns for Managing Evolution, and suggests that we can manage the structural and behavioural evolution of software by introducing a small number of novel design patterns. Unfortunately, these patterns, which essentially promote late (dynamic) binding, require proprietary language extensions (specifically for C++) and hence cannot be applied generally.

Mel O Cinneide's Ph.D. thesis [O Cinneide, 2000] prototypes a tool that assists developers in introducing design patterns into legacy systems. With this tool, the developer selects problematic areas of code and a design pattern he or she wishes to introduce into that code, and the tool helps introduce that pattern via a set of constituent mini-transformations (basically refactorings). Unfortunately, the tool only works for Java programs, and it currently only supports seven design patterns. Cinneide believes that, in principle, the tool could be extended to support up to 75% of the Gang-of-Four patterns [Gamma et al. 1995], at least partially, with the remaining 25% seen as essentially impossible to support. This work is clearly a valuable contribution toward patterns tools, and is hopefully a step towards user-extensible refactoring/re-patterning

tools, wherein support for new languages and refactorings/patterns can be added via configuration rather than via source code changes to the tool itself.

Research in Brazil [Masiero and Braga, 1999; Cagnin et al. 2000] suggests that there will probably already be many well known design patterns implicit in, albeit scattered throughout, the architecture of a given legacy system. Teasing out those patterns and making them explicit within the legacy system's architecture is seen as helping us better understand the legacy system's current behaviour in preparation for later reengineering work. Our own experience certainly confirms the validity of this approach, and we believe that refactoring (as described in an earlier chapter) would be the best way to tease these patterns out.

What doesn't come out of the Brazilian research, however, is that (at least in our experience) many of the patterns manifest in a legacy system are precisely those that we do not want to retain in a replacement adaptable system, since those patterns generally reflect the legacy system's current and, more importantly, wholly inadequate, commonality and variability assumptions. An adaptable system, then, needs to reflect instead patterns that embrace the commonality and variability inherent in the requirements that the legacy system currently resists. Thus, although it may be valuable to identify the current patterns within a legacy system, we have found in practice that we often only do so in order to understand where, when, and how they petrify the legacy system, in readiness for the prescription of appropriate antidotes to that petrification.

The FAMOOS team has developed an excellent handbook for reengineering [FAMOOS, 1999], including an interesting catalogue of reengineering patterns. Unfortunately, the handbook and patterns are deliberately targeted at already object-oriented systems, and hence have diminished value for legacy systems developed using other (typically pre-object) paradigms.

Closer to our own emphasis on patterns within the context of legacy systems is the work of Stevens, Pooley, and others at Edinburgh University [Stevens and Pooley, 1998; Dewer et al. 1999], and also that of O'Callaghan and colleagues at DeMontfort University [Dai et al. 1999]. Both sets of researchers offer their own catalogues of System Reengineering Patterns, and although there is some (very minor) overlap with our own pattern catalogue, the differences between their catalogues and ours are sufficiently great that we see this as confirmation of our finding (see below) that people really do need to "roll their own" pattern catalogues, to suite their own specific legacy system migration needs.

Interestingly, neither the Edinburgh nor the DeMontfort groups emphasise patterns that lead to petrification, instead focusing on solution-oriented patterns to achieve legacy system replacements. This is a surprising omission, since we have discovered during out own migration work that as legacy systems petrify there tend to be certain high-level recurring themes in that petrification, and recognising these patterns is an essential prerequisite to cultivating and prescribing appropriate antidotes [Lauder and Kent, 2000b; Lauder and Kent, 2001a].

13.4. Excited Flurries of Futile Activity

Unless we understand precisely why a specific legacy system is petrified we run the risk of clinging to imagined panaceas; enticed by the promise of alleviating the legacy systems headache with a simple pill, an all-in-one antidote. Unfortunately, simplistic universal cure-alls have proven time and time again at EDP to be dangerously distracting; resulting in overly excited flurries of blinkered activity. High expectations come crashing to a halt when temporarily forgotten, or ignored, or unidentified causes of petrification manifest in the legacy system at hand are eventually recognised as un-addressed by the current quick fix. All too often, the resultant delay leads to a state of ever-increasing panic thus raising the likelihood that further quick fixes will be grasped at in an attempt to catch up on lost ground. We have personally seen this boom to bust cycle spin for years on end.

Without reflection on the real problems actually present within a legacy system, we are in danger of rushing desperately towards futile quick fixes for intrinsically hard problems. As an example, one team we worked with leaped at the chance to tackle a legacy system using CORBA [Henning and Vinoski, 1999], which has been widely acclaimed as an excellent platform for distributed systems. Unfortunately, the lack of a distributed platform turned out to be the very least of the problems facing that particularly legacy system, and thus initial enthusiasm turned ultimately to disillusionment

13.5. Petrifying Patterns

We have learned, through practical participation in the research underlying this thesis, that this cycle of disappointment could have been avoided had those enticed by quick fixes been able to refer to cultivated catalogues of *Petrifying Patterns*[55] with which to

[55] Early in this research we termed them *anti-patterns*, following the lead of [Brown et al. 1998], but this term has become somewhat muddled in the software development community [Fowler, 1999a] and hence is best avoided.

evaluate the suitability of quick fix candidates. More specifically, a catalogue of Petrifying Patterns facilitates the identification of the range of problems that are actually present within a given legacy system, thus preventing an overly narrow focus upon whichever problem happens currently to be foremost in our mind. A candidate "fix" can then be evaluated in terms of its contribution to the resolution of the petrification manifest in the legacy system.

Our experience has been that this approach helps us keep legacy system migration on track by steering us away from distracting flurries of misguided displacement activity, refocusing our attention instead on the very real issues that need addressing.

13.6. Seven Deadly Sins

Amusingly, it appears that all "bad practices" in software development can be traced back to the seven deadly sins of haste, apathy, narrow-mindedness, sloth, avarice, ignorance, and pride [Brown et al. 1998]. Within the context of a specific legacy system, however, we need to be more specific. We need to uncover the actual Petrifying Patterns prevalent in the specific legacy systems we are dealing with.

13.7. Symptoms and Causes

We can think of Petrifying Patterns as *contra-indicated* patterns (in the medical sense that they are "bad medicine" for this particular patient). Petrifying Patterns tend to perpetuate various undesirable properties impeding the reflection of requirements change. Each of these undesirable properties exhibits a number of *symptoms* and each of those symptoms has underlying *causes*. As shown in Figure 13.1, it is the combination of these symptoms and causes across the prevalent Petrifying Patterns that underpins a legacy system's (petrified) accidental architecture.

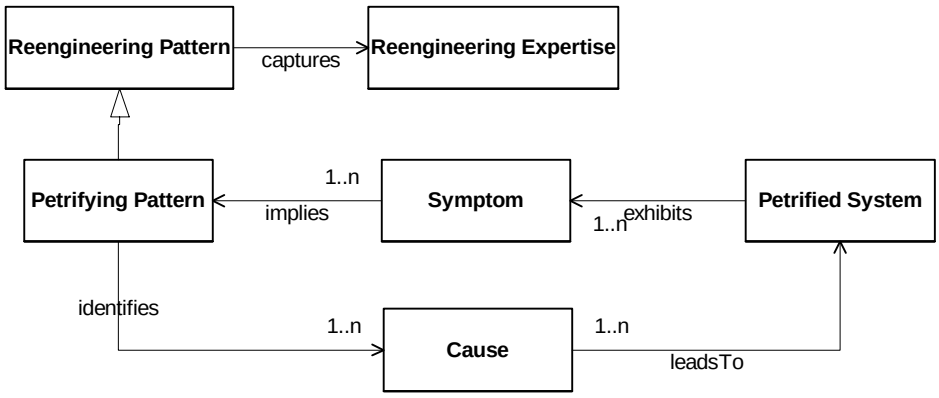

Figure 13.1 – Petrifying Pattern

By continually capturing Petrifying Patterns, we progressively gain reusable expertise with which to diagnose petrification problems in the future.

13.8. Productive Patterns

Having identified petrification problems in terms of symptoms and causes, we are then better positioned to prescribe solutions to those problems in terms of candidate *antidotes*. *Productive Patterns* are the medicine prescribed to relieve petrification. Productive Patterns capture *features* and *benefits* that are desirable in adaptable systems (see Figure 13.2).

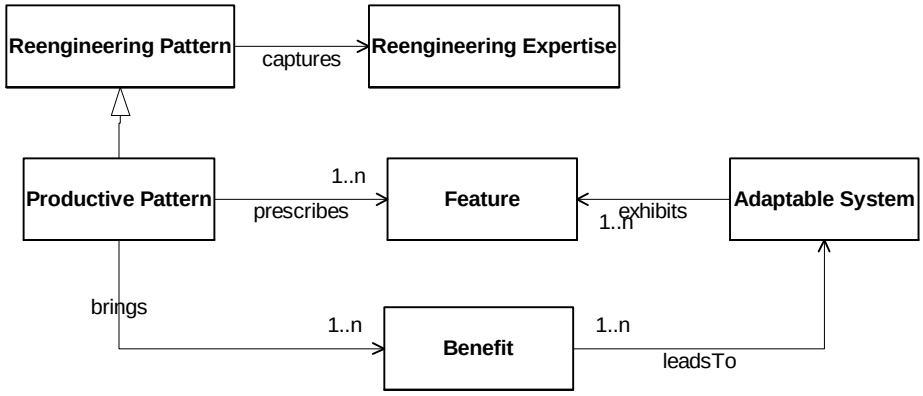

Figure 13.2 – Productive Pattern

Thus, throughout Productive Migration we first identify the symptoms and causes (Petrifying Patterns) actually present within a given legacy system, and then prescribe appropriate remedial features and benefits (Productive Patterns) as antidotes.

13.9. Perpetually Monitored Evolution

Identifying a system's present Petrifying Patterns and deploying corresponding antidote Productive Patterns in their place supports and inspires us in *migrating* a legacy system (during Productive Migration) from its accidental architecture to a more legitimate architecture. The challenge then shifts (during Productive Evolution) to a perpetual process of *monitoring the evolution* of a newly adapable system with an eye to nipping Petrifying Patterns in the bud, before they have a chance to take hold. Thus, we tackle Petrifying Patterns and prescribe Productive Pattern antidotes primarily during Productive Migration, since Productive Evolution is the healthy living that (hopefully) prevents such medicine from being necessary by steering us away from petrification in the first place.

13.10. Patterns Complement Refactoring

Petrifying and Productive Patterns are generally higher level than, and indeed complementary to, the small-scale refactorings discussed in earlier chapters. Both assist us when transforming a petrified system to a suitably adaptable replacement, but patterns can help us to think at a higher level of abstraction, seeing the "big picture" that can be lost in the details of small-scale changes. They help us to think strategically about petrification, whereas refactorings may be thought of as the tactics with which those strategies can be realised. Over the past couple of years employing Productive Migration, it has sometimes been our experience that developers get stuck in a migration "rut", wherein a succession of Bursts appear to be "treading water" addressing much the same small-scale concerns again and again. It is at precisely such times that appeal to a well-cultivated catalogue of patterns can lead to the inspirational "Aha!" that throws us out of the loop and into seeing some particularly thorny problem in a completely different way.

13.11. Balancing the Catalogue

Preliminary extracts from the catalogue of Petrifying and Productive patterns mined at EDP have been documented elsewhere [Lauder and Kent, 2000b; Lauder and Kent, 2001a]. A more comprehensive and up-to-date sampling is presented in the appendices. Early in this research, the focus was primarily upon uncovering technical patterns [Lauder and Kent, 2000b]. However, as the research has matured in response to ongoing observations of petrification, the catalogue has become more balanced in that it now reflects developer and management practices that are equally contributory to system petrification as technical concerns[56]. This confirms findings elsewhere that “the real barriers to success are frequently not technical, but are related to management and culture” [Bergey et al. 1997]. Indeed, it transpires that nine of the top ten reasons why reengineering projects fail are essentially to do with organisational management and culture [Bergey et al. 1999]. The remaining reason for reengineering failure is a technical one: “Software architecture is not a primary reengineering consideration” [ibid.], which confirms our own findings of the centrality of architecture to understanding and addressing system petrification.

[56] Ramage and Bennett would have us go even further here, pointing out that it is often not just the software system that is petrified, rather it may be the whole organisation since socio-technical concerns are frequently heavily intertwined with one another. We cannot, therefore, consider the petrification of software without also considering the petrification of the environment within which that software plays a part [Ramage and Bennett, 1998].

13.12. Let the Punishment Fit the Crime

Initially, we advocated a simple, universally applicable mapping from each Petrifying Pattern to a corresponding antidote Productive Pattern. We recommended that developers delve into their legacy systems in search of lurking instances of our catalogued Petrifying Patterns and immediately eliminate the culprits via their replacement with our magical remedies.

Experience has taught us that this simple mapping strategy will certainly succeed in eliminating real problems within legacy systems. Unfortunately, we have found that it also tends to identify false problems and prescribe unnecessary antidotes that introduce problems of their own. As an example, in the appendices we have documented Tight Coupling (p.207) as a Petrifying Pattern, the antidote to which is Implicit Invocation (p.246). With Tight Coupling, software components are aware of each other's existence, maintain explicit links to one another, and invoke one another's operations directly. This results in intricate interdependence between the components (the *cause*), which leads to a cross component fragility where changes to one component may impact another component (the *symptom*). With Implicit Invocation, components are de-coupled via an intermediate broker, resulting in loose coupling (the *feature*) wherein inter-component dependencies are minimised thus allowing components to be modified without fear of side-effect impacting on other components (the *benefit*).

The problem with universal mappings, such as that from Tight Coupling to Implicit Invocation, is that they do not take context into account. We cannot say whether a given pattern (just like a medicine) is contra-indicated or is an antidote without first qualifying the context within which that declaration applies. A given pattern may be an antidote in some circumstances, and contra-indicated in others. Sometimes, the treatment can be more damaging that the ailment it cures.

For example, in cases where two components are inherently closely related, requiring intimate knowledge of one another, de-coupling could introduce more problems that it solved. In cases like this, Implicit Invocation would, in fact, be a Petrifying Pattern, and Tight Coupling its antidote! Every pattern, then, has tradeoffs, and no pattern is a “silver bullet”. It is extremely naïve to apply a pattern unquestioningly.

13.13. The First Law of Medicine: Do No Harm

When evaluating the suitability of a given Petrifying Pattern to diagnose a particular case of petrification we need to look not just at what symptoms that pattern *could* cause, but more importantly, which of those symptoms it *actually* causes in a given legacy

system. Petrifying Patterns whose symptoms are imagined rather than manifest within a given legacy system should be left well alone. For example, it took us a long time to realise that not every “dumb terminal” application was suffering from the Ugly Duckling Petrifying Pattern (p.214).

To differentiate between real and imagined problems, we need to delve deeper. Thankfully, Productive Migration gives us some guidance here, and this is important: a given Petrifying Pattern is only present if it prevents the introduction of some specific and high-priority business value we need to add to the system under migration (see Figure 13.3).

Since Runs (and ultimately Sprints and Bursts) focus upon introducing specific business value, it must be left to the members of each Run (and Sprint and Burst) team to recognise which Petrifying Patterns (if any) prevent the introduction of that business value and which (if any) of the candidate antidote Productive Patterns can help eliminate the aspects of petrification which prevent its introduction.

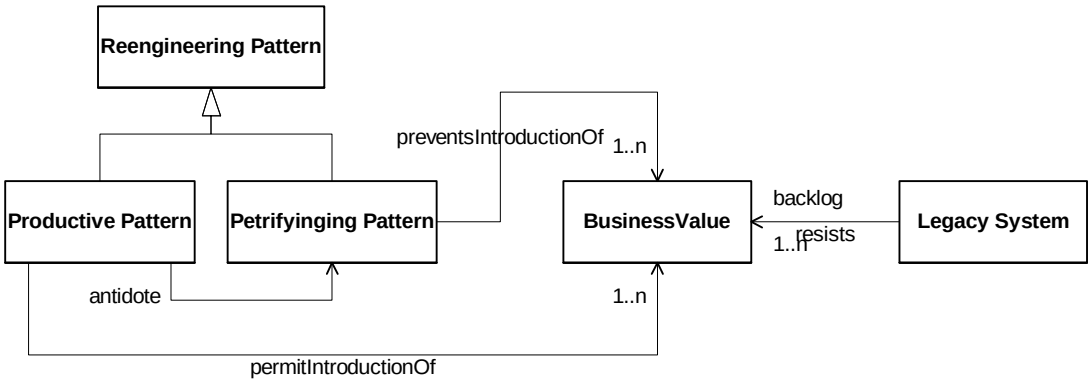

Figure 13.3 – Resisted Business Value Drives Productive Pattern Prescription

13.14. A Catalogue is Not an Expert

Note that a pattern catalogue is only a resource: it is not in itself an expert to be deferred to unquestioningly. A pattern catalogue is a means of acquiring expertise, not a substitute for it. This is particularly problematic, we have found, when using other people’s pattern catalogues (and a major incentive to cultivate our own): the original authors may not be available for us to benefit from their expertise and guidance directly.

13.15. A Pattern is a Metaphor

Our experience is that the expertise captured within a given pattern has to be acquired personally by reflecting deeply upon and immersing oneself within that pattern’s relevance and implications in individual and concrete problem contexts.

More specifically, an important realisation of ours from working with patterns is that a pattern reflects connotations, not denotations. That is, an effective pattern is generally highly metaphoric [Lauder and Kent, 2000a]. To be effective, then, a pattern cannot simply be "read" in the normal sense; they are not rigid, absolute texts to be followed without question. Rather, as depicted in Figure 13.4, a given pattern needs to be seen only a starting point, acting as a deeply inspirational "conceptual bridge" [Black, 1979] [Petrie, 1979; Shiff, 1979] that guides our personal unravelling of its diverse allusive ties[Coplien, 1998c] in the construction of rich, evocative, and personal mental models [Green, 1979]. Continual situated (i.e. in-context) elaboration, then, adapts a pattern to the particular needs of individual contexts against the personal background knowledge of the individual pattern "appreciator".

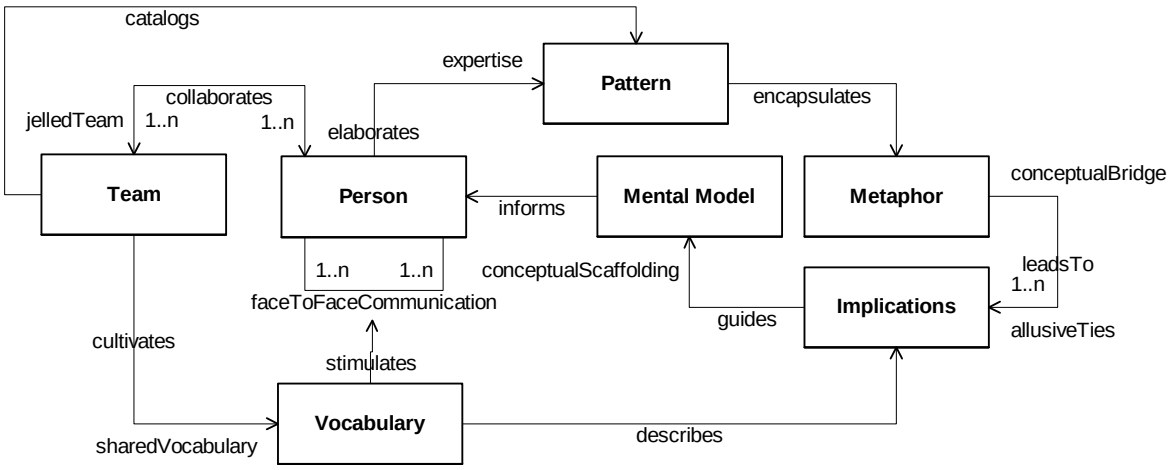

Figure 13.4 – Metaphors Stimulate Pattern Elaboration

13.16. Common Vocabulary

From working with patterns we have found that pattern elaboration is also often a group activity, albeit generally implicitly so. Much of the inspirational power of a pattern comes from the vocabulary inherent in the metaphor underpinning the pattern [Sticht, 1979]. This vocabulary starts with the pattern name, and proceeds though a hierarchical layering of constituent parts, weaving together the relationships (the metaphor's allusive ties) between those parts into a common whole. As people start to use this (common) vocabulary it begins to form a "conceptual scaffolding" [Veale, 1995] enabling pattern appreciators "to discuss, document, and explore design alternatives at a high level of abstraction … [raising] the level at which [they] design and discuss designs with [their] colleagues." [Gamma et al. 1995]. As we appeal to a pattern's vocabulary within a specific context, we are stimulated (guided) by the pattern to follow (elaborate) its

allusive ties in the directions most appropriate for a suitable tailoring of the pattern to the needs of that specific context.

13.17. Pattern Elaboration Families

An interesting outcome of our ongoing work with patterns is that elaboration and deployment of patterns in new and diverse contexts leads to an ever-increasing family of cultivated elaborations. As membership of an elaboration family increases, each family member constitutes a possible starting point for yet further elaborations. The elaboration of a given family member, in context, has revealed to us fresh insights whose scope extends beyond that specific context and to the pattern in general. Consequently, as depicted in Figure 13.5, the revelations of a given elaboration have the potential to ripple up and down the pattern's family tree, further elaborating, and hence enriching, other family members [Lauder and Kent, 2000a].

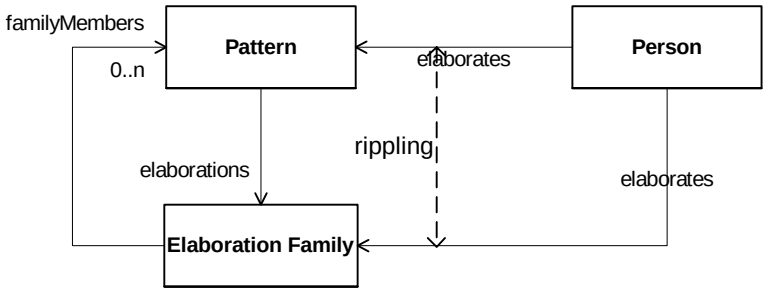

Figure 13.5 – Pattern Elaboration Families

13.18. Growing Expertise

It is this continual experience-led process of the population of families of pattern elaborations that transforms a pattern novice into a pattern expert. In other words, we found that if we are to become pattern experts we need to capture not just the patterns themselves, but also our personal experience using them - our growing expertise.

13.19. Pattern Formalisation

There has been an interesting and ongoing debate with respect to whether or not formalisation of technical patterns is achievable and, more importantly, whether or not it is appropriate (e.g. for: [Eden et al. 1998; Wills, 1998], against: [Brown, 1998; Coplien, 1998a]). We constantly flipped between these two stances until we eventually realised that the debate is, at heart, a re-hash of earlier literalist/figuralist debates from the metaphor literature (see [Veale, 1998] for a good overview of the metaphor debates). An extreme formalist would adopt the literalist stance that a pattern is simply a shorthand expression of a more concrete design. The fear of the figuralist, on the other

hand, would be that formalisation of patterns leads to a frozen literal interpretation thus killing off the potential for the essential characteristics of ongoing active interpretation and application in unforeseen contexts that underlie pattern reuse. Any attempt to capture patterns in a formal way must not sacrifice openness of interpretation for the sake of precision.

We resolve this dilemma [Lauder and Kent, 2000a] by recognising that formalisation has the potential both to enable us to specify precisely patterns at the root of family trees and to record with equal precision, and distinguish unambiguously between, the family members resulting from ongoing elaborations. Precision would, therefore, enable us to identify and select with confidence the family member most appropriate to our current problem context or to recognise that the current problem context calls for the addition of a new family member via further elaboration.

13.20. Precise Visual Notation

We have begun to investigate a notation with which to express patterns both precisely and visually [Lauder and Kent, 1998], via a Constraint-Diagram-based [Kent, 1997] extension to UML [Rumbaugh et al. 1999] which makes a set-theoretic appeal to constrained meta-models. Note that we shy away from claiming that we can capture patterns formally with this notation, at least in the strictest sense, since although we are confident that we can express patterns with a high degree of precision, which we see as one of the main aims of formalisation, we have not attempted to construct a formal semantics for the notation to which we appeal.

As an example of the notation in use, Figure 13.6 below (taken from [Lauder and Kent, 1998]) captures the SetState method of the well known Observer [Gamma et al. 1995] design pattern. The basic idea in Figure 13.6 is to express an operation in terms of pre- and post-conditions (depicted as Constraint Diagrams) connected via a (modified) UML Sequence Diagram to depict operational dynamics. Importantly, we shy away from concrete prototypical instances of methods, data members, and objects (as used in, e.g. [Gamma et al. 1995]), since these are overly constrained and hence do not depict a pattern accurately. Instead we reveal aspects of the pattern's meta-model (which describes the semantics of concrete models), through abstract sets of properly constrained methods, data members, and objects. Thus, in Figure 13.6, circles represent sets, dots within circles represent set members, and squiggly lines between set members indicate potentially shared identity. We also add a fourth compartment to UML class

diagrams to depict abstract instances (objects) of that class, circumventing the need for overly constrained (prototypical and concrete) object diagrams.

Admittedly, Figure 13.6 is rather involved; it is intended only to give a taste for the notation and principles behind precise pattern expression, each of which are more thoroughly described in [Lauder and Kent, 1998].

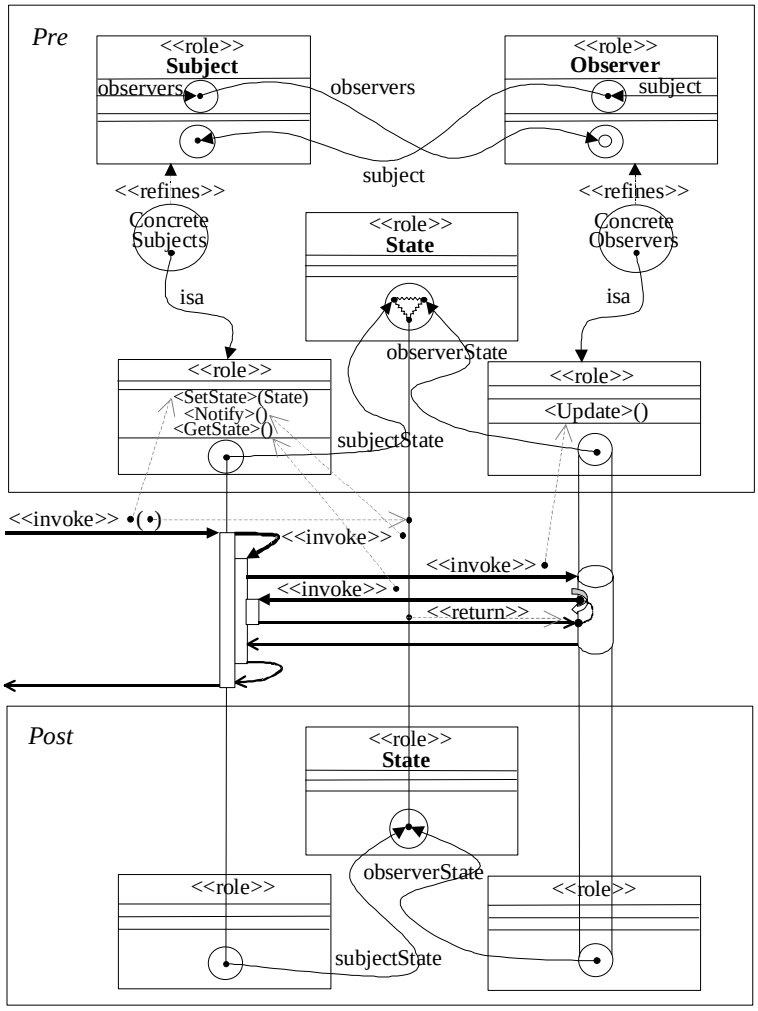

Figure 13.6 – Formal Specification of Observer::SetState

Having worked with our notation to capture patterns precisely, we have discovered that to completely express a pattern in this way typically requires page after page of diagram, each as intricate and verbose as Figure 13.6. Consequently, paperbound elaboration of a pattern catalogue soon becomes a maintenance nightmare. Unfortunately, tool support to assist cataloguing, elaboration, and deployment of such pattern models lags far behind the notation. Certainly much work remains to be done in this area. However, our initial optimism for the emergence of suitable tools in the near future has been somewhat tempered by recent discoveries with respect to the up-take of high-end software tools (see chapter 14).

13.21. Feeding Rich Mental Models

Specifying patterns precisely (or even formally), then, does no harm; indeed, it may well do some good. Whether or not we express patterns in this way, though, the main message to come out of pattern use is that precisely specified models of patterns are of little use unless we ultimately absorb those precise specifications into rich and evolving mental models with which we can explore and elaborate the patterns effectively.

13.22. Misdiagnosis and the Wrong Prescription

We have learned that an experienced migration team who understand their personally cultivated catalogue of Petrifying and Productive Patterns deeply will find that the patterns themselves can help prevent both under-ambition and over-ambition when performing migration work. Put simply, a cultivated pattern catalogue, when used intelligently, can help keep a legacy system migration project on the straight and narrow. In particular, an assimilated pattern catalogue can guide the team and help them to recognise and thus avoid at least five forms of mismatch between the needs of the legacy system and the prescribed course of treatment:

1. **Wrong Illness:** this form of mismatch is where a condition is considered contra-indicated when, in fact, that aspect of the legacy system is perfectly healthy. This identification of false problems can lead to very damaging unwarranted treatments. We have personally seen an astonishing example of this at EDP, where dates were stored in a file in one format (day count from a given base) whereas reports required their display in another format (dd/mm/yy). This is a very commonplace design, and certainly not a contra-indicated pattern at all. Storing dates in numeric count format allows simple date arithmetic and data dictionaries are typically used to specify that, when printed, the date should be converted to a different format. Unfortunately, a naïve programmer decided that this date mismatch was a "bug", and implemented what he considered appropriate remedial action. His “fix” traversed the whole file changing dates from count format to display format in-situ, then printed the required report, and finally traversed the file again converting all dates back to count format. It was only the keen eye of an experienced developer that stopped this code from shipping. In this case, a pattern catalogue may have alerted the junior programmer to the fact that expertise with such "problems" should be recorded. Appeal to a catalogue would have shown his "problem" to be novel, thus requiring its discussion with experienced developers before adoption and inclusion in the catalogue.

2. **Pill-Popping:** this form of mismatch is where an antidote is applied indiscriminately on the assumption that it can do no harm and its "better to be safe than sorry". One example we have seen of this is where the Dynamic Pluggable Factory pattern [Lauder, 1999c] (and p.229), which adds new classes to a system at runtime, was applied to the very heart of a large application. This pattern was introduced on the whim "who knows, one day we might have new kinds of objects coming in all the time". It transpired that even though the particular application did require new classes of object on occasion, it was never really Bound-and-Gagged (a Petrifying Pattern wherein the available classes are too limiting, see p.186). The frequency and urgency of requirements for new classes was so low that it almost certainly did not warrant the complexity trade-off inherent in Dynamic Pluggable Factory. In this case, it would probably have been better to apply no treatment at all, since there was no significant problem needing an antidote.
3. **Treatment of Benign Conditions:** this form of mismatch is where the proposed Productive Pattern addresses a Petrifying Pattern, which, although present, is not actually detrimental to the particular legacy system under our care. For example, at EDP there was a backlogged request to migrate a system-administration tool to multiple platforms (Windows, Linux, and Apple Macintosh) when in fact it actually transpired that all targeted users were only interested in a Windows-based solution. Subsequent appeal to our pattern catalogue allowed those involved to recognise that, although this particular software product exhibited Ball and Chain (wherein the system is tied to a specific platform, p.181), it was inappropriate to apply an antidote since Ball and Chain had no detrimental effects in this particular case. On this occasion, then, we were able to prevent unnecessary work and direct development effort elsewhere.
4. **Absence of Holistic Medicine:** this form of mismatch is where the proposed treatment addresses only some of the present Petrifying Patterns and where other issues remain un-addressed which are at least as pressing. This is not necessarily a problem in itself since other antidotes can be applied to address the remaining issues. However, there is often considerable benefit to be gained from reviewing whether or not multiple present Petrifying Patterns can be tackled in unison. At the very least it is sensible to ensure that effort expended in the resolution of the first set of Petrifying Patterns to be tackled is relevant to the resolution of further Petrifying Patterns. This is where Marathon and Run steering sessions are invaluable: if we relied solely on Sprints and Bursts we would risk being unable to see the wood for

the trees. We have seen an example where preliminary work to bring a Microsoft Windows GUI to an application (addressing Ugly Duckling, p.214) was later abandoned since it conflicted with the need for an HTML front end (used, in-part, to address Human Shield, p.195). Earlier recognition that the application would require connection to the Internet would have allowed us to kill two birds (Ugly Duckling and Human Shield) with one stone (an HTML-based interface).

5. **Major Surgery for a Minor Complaint:** this form of mismatch is where the antidote is overkill. The corollary to the previous example is an ambitious attempt that we have witnessed to completely "web-enable" a large-scale application despite the overlooked fact that few if any users required anything more than a Windows front end. The application in question certainly suffers from Ugly Duckling but to apply anything more substantial than a Windows wrapper would seem to be an inappropriate diversion of effort from more pressing concerns. In this case our pattern catalogue enabled managers and developers to recognise that the solution in hand was analogous to "cracking a nut with a sledgehammer" and hence the project was scaled back to a more appropriate scope.

13.23. Review

Productive Migration and Evolution, as described in earlier chapters, should be seen only as a starting point. To make them useful (and to prevent them from petrifying themselves) we need to continually adapt these processes to our evolving needs, to evolve our own body of reusable migration and evolution expertise, and to understand when to apply that expertise and when to look elsewhere. Appeal to cultivated catalogues of Petrifying and Productive patterns will help us to do just that.

We have seen in this chapter that such patterns are deeply metaphoric, and undergo elaboration into rich families of related patterns, each of which may then form the starting point for further elaboration. As we elaborate pattern families, we may choose to capture family members precisely. We have experimented with possible notations for doing just that, although much work remains to be done in the area, not least in terms of adequate tool support to assist pattern cataloguing, elaboration, and deployment.

It is important to recognise that the Petrifying and Productive patterns detailed in the appendices are mined specifically from the legacy systems that were investigated. Although we are confident that many of these patterns will have relevance to other projects, they are unlikely to be universally valid. The appropriate catalogue of contra-

indicated and antidote patterns will vary from project to project, since each project will vary in terms of the prevalent symptoms and causes of, and resolutions to, petrification.

Thus, although the patterns documented here may well be applicable to other legacy systems, it is essential for each organisation to cultivate its own pattern catalogues. Each organisation - probably even each project - needs to capture both their own experience with the very real problems facing them and proven resolutions to those problems, rather than pinning hopes on speculated concerns from other projects. A cultivated and personalised pattern catalogue continues to capture and evolve with its caretakers' own migration experience, growing with them, and adapting with their changing needs. Not only are self-cultivated pattern catalogues likely to be more relevant than patterns originating elsewhere, but the migration team will understand them more deeply since they originate in the team’s own experience. From this deeper understanding comes a greater awareness of when the patterns are applicable and when they will do more harm than good.

Thus, a cultivated catalogue of personalised patterns helps us to achieve a living, learning, agile process, co-evolving with our migration and evolution expertise. This will ensure that we are better prepared for the petrification challenges that are yet to face us than we have been for the legacy systems that have held us back thus far.

Chapter 14

Looking Ahead

Let's want no discipline, make no delay;
For, lords, tomorrow is a busy day.
- William Shakespeare, *Richard III* V, 3

14.1. Preview

So, how do we take this thesis forward? An "obvious" next step for this research would be to investigate tool support for legacy system migration and evolution. In this penultimate chapter we dampen initial enthusiasm for tool support by observing that, at least in the experience of our industrial collaborator, current automated tools are often more hindrance than help. We then turn to asking what future tools would need to look like to be useful to legacy system migration and evolution experts. It transpires that to find the answer we need to turn to the experts themselves.

14.2. Tool Support

Many research projects seem attracted to prototyping some software engineering tool[57], and this project is no exception. As noted in the previous chapter, this project began with considerable enthusiasm for precise pattern specification and deployment. It was envisioned that some prototypical tool might be developed to demonstrate the feasibility of automated assistance for precise visual pattern cataloguing and application during legacy system migration work.

14.3. Smirks and Raised Eyebrows

Unfortunately, and here is the rub, when the possibility of such tool support was discussed with our industrial collaborator they just didn't seem to care. Here are a few captured comments:

"Tools are okay if you can work the way the tool wants you to work"

"We've used [an impressive catalogue of tools] and they just slowed us down"

"You've got to be light on your feet around here, twiddling with the code, not drawing pretty pictures"

"The only tools we need are vi, grep, diff, and the compiler"

[57] See [CASE, 2001] for an index to literally hundreds of such tools.

In summary, EDP's considerable experience with automated tools left them jaded[58]. Any mention of tool support was constantly greeted with smirks and raised eyebrows. In response to their lack of enthusiasm, our research shifted to asking the people at EDP what they thought would help them with their legacy systems, and their responses (i.e. working out what's wrong with them, and how we can learn to sort them out) were the underlying motivation for the work whose findings are described throughout this thesis.

14.4. Losing Control

Throughout the duration of this work there has grown a nagging feeling that there may well be something to what the developers at EDP are saying. Having "played" with many of the tools available to them, it is certainly hard to see how any of the tools would have made any of EDP's projects more successful. Of course, ego could (and admittedly did) draw us into thinking "ah, yes, but the tools they had weren't good enough. Let's make a better one", and this is when the eyebrow raising began.

EDP's scepticism seems to boil down to the fact that, at least in their experience, "big" tools don't seem to cut the mustard in practice. The term "big" is used here to denote the commonplace practice where the tool tries to guide the developer in some way, and thus doesn't allow the developer to remain in control. For example, experience with a major UML-based modelling tool was seen as trying to enforce a particular form of bureaucratic model-driven development. Similarly, experience with a popular refactoring browser was seen as artificially forcing the programmer away from the code. As a final example, experience with a certain integrated development environment mandated acceptance of the tool's implicit assumptions about project structure.

14.5. Opportunistic Style

Reflecting upon all this, and watching developers in practice throughout this project, it seems clear that as they migrate legacy systems and evolve adaptable systems they need to be highly opportunistic, employing many diverse strategies throughout development as opportunities arise and needs dictate. Furthermore, we have noticed that developers tend to switch between these diverse strategies very rapidly.

Rather than appeal to a tool that enforces "one right way" of doing things, then, the programmers we have worked with turn instead to a diverse collection of small, fast, and highly focused tools with which they are intimately familiar and adept at flipping

[58] Compounding the problem, of course, as [Pigoski, 1997] (building on [Sharon, 1996]) observes, may be the fact that most CASE tools have a strong bias toward new development work, and give little emphasis to the "maintenance" of systems that already exist.

between very rapidly. That is, even with impressive discretionary budgets, and shelves full of high-end tools[59], the EDP developers still appeal in their daily work to a mix of old-time favourites: a simple text editor, a compiler, a search tool, a file copier, and so on. High-end tools, by taking control away from the developer, stifle the opportunistic pick-and-mix, on-the-fly, style that these smaller tools enable.

14.6. Remaining Optimistic

Interestingly, despite strong scepticism at EDP with respect to "big" tools, there still remains a backdrop of optimism with respect to the potential value of future tools, so long as those tools allow the programmer to remain in control. For example, existing refactoring browsers force specific refactorings (and even specific languages) on the developer. Developers at EDP appear receptive to the idea of a refactoring browser wherein the developer is able to institute their own refactorings (and possibly diverse programming languages) into the tool. More importantly, it is seen as essential that such a browser does not require the developer to be "in" some refactoring tool environment to apply the refactorings, rather the refactorings need to be applicable on-the-fly "anytime, anyplace, anywhere". How on earth such a tool would work is entirely unclear, but without such investigation it seems that many potential tools will remain largely unused.

14.7. Validation

An impressive survey of reverse engineering and reengineering tools by the Software Engineering Institute [Reengineering Center, 1995] appears to validate our experience in this area. The SEI report found all of the surveyed automated tools to be lacking. In particular, the authors of the report discovered that tools tend to work at a different level to the developer. The developer is constantly trying to evolve and utilise high-level domain-oriented mental models whereas tools are forced to work without background domain knowledge and hence can recognise only low-level solution-oriented objects in code. More recent work by Arie van Deursen and colleagues, in their ongoing attempts

[59] These products stay literally "on the shelf". There is evidence that this experience is commonplace; certainly a great many software tools are purchased, resulting in a booming industry, but once purchased many are rarely used in practice [Pigoski, 1997] (quoting evidence presented in [Vollman and Garbajosa-Sopena, 1996]). On the other hand, [Pigoski, 1997] also contends that in many cases this may be due to a reluctance to accept the substantial training investment generally required to benefit from these products, and also the uncomfortable fact that, although such tools may improve quality, there is evidence [Jones, 1991] that they reduce productivity as a side effect. With considerable commercial pressures to get "good enough" [Bach, 1997] software to market quickly, it seems likely that many organisations are unwilling to accept this trade-off.

to construct tool-based legacy system “renovation factories”, confirms this finding [van Deursen et al. 1999].

Furthermore, in the context of legacy systems, much relevant information is scattered throughout diverse fragments of code and tools are hard pressed to recognise their relatedness. Thus, the tool is pushing out views that are at odds with the mental model the developer is building.

14.8. Novices vs. Experts

In summary, the SEI report found that high-end tools are useful only to novices who lack any real domain knowledge and thus must appeal to an essentially bottom-up domain-learning strategy by systematically ploughing through the tool’s view of the code.

Experts, on the other hand, were found to adopt a highly opportunistic mixture of bottom-up, top-down, and, indeed, middle-out strategies wherein they continually apply their substantial base of domain knowledge to “fill in the gaps” in their understanding of how the system hangs together. Furthermore, it was found that as systems become larger and more complicated (typical of legacy systems), it becomes absolutely essential to employ this type of mixed approach since a bottom-up-only strategy (the novice approach) leads eventually to an intellectually unmanageable swamping in details that the developer cannot possibly remember.

14.9. Size Matters

The SEI report authors, then, treat “big” tools with caution. As they point out, the effectiveness of such tools tends to be demonstrated in studies that deal only with relatively small code samples. If this is true for commercial products, it is even more so for research prototypes. The problem is, the value demonstrated when programming in the small rarely scales up to large systems spanning several million lines of code. This was precisely the problem facing EDP; tools that worked well in small-scale demonstrations were completely ineffective when faced with five million lines of scattered, petrified code.

14.10. Learn from the Experts

So, what types of tool will actually be useful to the developer working with legacy systems? Turning again to that SEI report: “One strategy for identifying appropriate maintenance tool support is to begin by classifying maintenance along with the strategies that have proven effective in completing these activities. From this

classification and set of techniques, specific tool needs will likely become apparent" [Reengineering Center, 1995].

In other words, we need to keep capturing patterns of migration and evolution expertise. We need tools that support that expertise and that can evolve and grow with it.

14.11. Review

"A bad workman blames his tools," but a bad tool can hinder a good workman. Expert developers need many diverse tools in their toolkit, since they employ many diverse strategies throughout their working day. Until we better understand how experts work when dealing opportunistically with legacy system migration and evolution, we can't expect to create tools that are truly helpful to them in that work. We need to keep learning from the experts themselves, to continually capture their patterns of reengineering best practice. Only then can we hope to build tools that help the experts to continue to work expertly.

Chapter 15

Contributions

Let us from point to point this story know,
To make the even truth in pleasure flow.
- William Shakespeare, *All's Well That Ends Well* V, 3

15.1. Preview

In this final chapter we first justify the thesis title, and then lead into a brief recap of the particular contributions to knowledge contained within this thesis.

15.2. Grand Title

The original, and rather grand, title for this research project was "Precise visual patterns for the evolutionary migration of legacy system to reusable components". Throughout this research, it has become evident that it is indeed possible to capture patterns, some of them very specific [Lauder, 1999c], both precisely and visually [Lauder and Kent, 1998; Lauder and Kent, 1999b], and we can certainly appeal to them for the evolutionary migration of legacy systems [Lauder and Lind, 1999] to reusable components [Lauder, 1999a; Lauder, 1999b; Lauder and Kent, 1999a].

15.3. Pill Popping

The results from this original strand of research certainly proved quite rewarding, at least in an intellectual sense. Unfortunately, addressing this research in a commercial context, rather than an entirely academic one, has been rather sobering. It has transpired that, although the project's initial title is pleasantly "buzz-word compliant", it rather misses the point. More specifically, upon reflection, it is now clear that the original project title was rather value laden, in that it prejudiced the early part of the research towards entirely speculated solution-oriented directions without first determining the actual problems underling real legacy systems. Indeed, perhaps the major eventual finding of this research has been that we really needed to uncover the real problems underpinning legacy system petrification before prescribing appropriate antidotes [Lauder, 1999d]. With hindsight, the presuppositions in the original title now reek of pill popping: throwing solutions at a patient hoping to catch whatever illness they may have.

15.4. Particular Contributions

Throughout the course of this research, then, the title for this thesis has evolved, quite radically, to “A Productive Response to Legacy System Petrification”, which first focuses more appropriately upon why legacy systems are problematic (i.e. denial of unanticipated change, and hacking-induced petrification) and only then upon how best to tackle them (i.e. boosting business-value-oriented productivity, and capturing and cultivating patterns of “best practice” [Lauder and Kent, 2000b; Lauder and Kent, 2001a; Lauder and Kent, 2000a]). In this context, then, the particular contributions of this thesis are seen to be:

Chapter 2: Appeal to exchange theory and the language action perspective to explain the need for, and the shape of, continual business process change.

Chapter 3: Redefinition of the term legacy system as an information system for which “the business processes it is intended to support are changing at a rate that exceeds the rate at which the information system can itself be adapted to reflect those changes”.

Chapter 4: Introduction of the concept of petrification, which encumbers systems, through disgraceful evolution, with change-resistant accidental architectures.

Chapter 5: Enumeration of three levels of developer maturity and recognition that developers need to work at the Adaptation level to respond to petrification effectively.

Chapter 6: The discovery that predictive (application, and in particular architecture, oriented) methodologies mitigate the risk of anticipated change at the expense, and even denial, of more-problematic unanticipated change, thus encouraging petrification.

Chapter 7: The discovery that appeal to the principle of embracing (unexpected) change within the context of a lightweight agile methodology can both prevent petrification and address systems that are already petrified.

Chapters 8, 9, 10, & 11: Introduction of Productive Migration as an effective methodology with which to address system petrification by productively and incrementally migrating legacy systems to adaptable replacements.

Chapter 12: Introduction of Productive Evolution as a continuation of Productive Migration to ensure that newly adaptable systems never again petrify into change resistant legacy systems.

Chapter 13: Assertion that we need to shy away from futile cure-alls, with the discovery that cultivating and appealing to personal catalogues of Petrifying Patterns pin-points the symptoms and causes of petrification manifest in a given legacy system,

so that the prescription of appropriate Productive Pattern antidotes is well-grounded in the needs of that legacy system.

Appendices: Introduction of a partial catalogue of both Petrifying and Productive Patterns mined and refined through empirical study, as an inspirational starting point for the elaboration of similar catalogues within the context-specific requirements of other legacy system migration and evolution efforts.

15.5. Parting Words

All of the ideas underlying the Productive Response to Legacy System Petrification, as presented in this thesis, have been tried out and refined by us "in the field". The results, of course, are preliminary, having been derived only over a relatively short timeframe (approximately three years) and specifically restricted to one organisation (albeit one with multiple subsidiaries, and several large legacy systems).

Although the author was deeply involved personally in migration projects at EDP, and saw firsthand the positive impacts of Productive Migration and Evolution on those projects, much of the feedback from other participants at EDP was primarily anecdotal (albeit substantiated by various sources within the company). For example, one group of developers abandoned a particular migration effort after two years of "getting nowhere" using traditional techniques. A second group restarted the same effort, this time following the practices outlined in this thesis. This second team was ultimately widely acclaimed for delivering a "superior solution" within just over a year. The second team, then, was certainly seen as more productive than the first (in terms of delivering higher business value in a shorter timeframe).

It would be nice to credit Productive Migration and Evolution alone for these improvements. Unfortunately, it is impossible in such a small study to rule out the potential impact of other factors such as difference in team ability and the fact that the second team was able to reflect on the mistakes of the first. Nevertheless, discussions with members of the second team indicate that they believe wholeheartedly that Productive Migration and Evolution contributed greatly to their productivity improvements.

Anecdotal evidence, however, is only partly conclusive. Upon reflection, one minor disappointment at the end of project is that, due to its limited scope and size, it has been impossible to collate any significant numerical metrics with which to back-up anecdotal praise. For metrics to have statistical validity, however, the results described here would need to be reproduced in a much larger context, with many more organisations and

many more legacy systems than could possibly have been included in our own study, given the limited resources available to us. It would be extremely gratifying, then, if other research projects and commercial organisation picked up where this project ends, and attempted to provide precisely that larger context.

For other organisations, then, it is hoped that this thesis will prove a sufficient starting point for them to “get the ball rolling”. However, for the thesis itself to not become a “petrified legacy system”, there needs to be an ongoing commitment to cultivating a personalised interpretation and refinement of the recommendations contained herein, against the backdrop of ever-changing contextual needs.

I hope people will play with the ideas presented here. Working on this thesis has completely changed the way I think about software systems, and more importantly, it has completely changed the way I now go about developing them. I hope it can influence others in the same way.

Appendix A

Petrifying Pattern Catalogue

And now remains
That we find out the cause of this effect,
Or rather say, the cause of this defect.
- William Shakespeare, *Hamlet* II, 2

Petrifying Pattern Form

Pattern Name

Name: Pattern Name

Type: Petrifying

Summary: A brief overview of the essence of the pattern.

Antidote: The productive pattern that is the primary antidote to the underlying causes of petrification and hence eliminates the symptoms of this pattern.

Symptoms: Indicators that this pattern may be present in the system under investigation.

Causes: Possible underlying causes for the symptoms outlined above.

Impact on Legacy Systems: How this pattern contributes to the petrification of a legacy system.

Known Exceptions: Instances when the presence of this pattern may not actually be deleterious.

Example: A concrete example of the existence of this pattern in a real-world system, and the negative impact it had on that system.

A.1. Accidental Architecture

Name: Accidental Architecture

Type: Petrifying

Summary: Where support for business processes (i.e. requirements) is scattered throughout and submerged within the details of implementation code it becomes difficult to construct mental models that trace between the requirements and the code. Thus, it becomes difficult to map ongoing requirements changes to appropriate changes in the implementation.

Antidote: Legitimate Architecture (p.254)

Symptoms: Information systems support business processes. The developers of an information system need to be able to reflect changes to business processes in the information systems supporting them. When it is difficult to understand exactly how an information system implements support for business processes ("I just can't find the code that's doing it"), or indeed what those business processes even are (see also the Gold In Them Thar Hills petrifying pattern, p.192), and we become overwhelmed when following trails of scattered code ("Its going to take ages to get my head around this code"), we may well be suffering from an Accidental Architecture. Another symptom is likely to be where there is a considerable lack of proximity between the scale of a requirements change and the effort anticipated to realise that change in code. Particularly troublesome here is that when we cannot understand the code, and cannot relate it to requirements changes, we become fearful (i.e. human petrification) of the potential side effects of any changes we may make. Thus, we either shy away from making those changes, or force fit (hack) them inappropriately into the architecture, which compounds the problem.

Causes: Support for business processes tends to be scattered throughout and submerged in low-level details of an information system's implementation. This scattering and submerging of requirements reflection may be the result of poor up-front planning when a system was originally developed. More likely, though, is that a system supported its original requirements fairly cleanly, but ongoing requirements change has upset the commonality and variability assumptions underpinning the current architecture, and a progression of minor "tweaks" (hacks) were each innocuously force-fitted into the existing design. The cumulative effect of these hacks was an eventual submerging of any directly traceable support for requirements. Hacking, then, layers an implicit

(accidental) architecture on top of the information system's legitimate architecture. Successive waves of hacking, in response to ongoing requirements change, introduce layer upon layer of implicit architecture into the information system. Eventually, comprehension of the architectural implications of these inter-dependent implicit architectural layers overwhelms the intellectual capacity of information system developers to the point where they are unable to anticipate and manage the side effects resulting from further change.

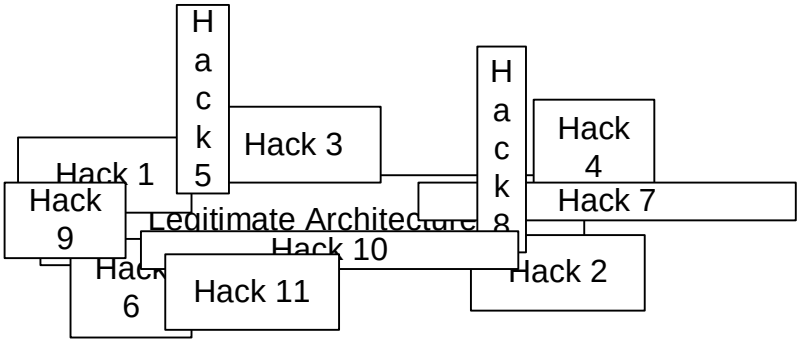

Figure A.1.1 – Accidental Architecture

Impact on Legacy Systems: When we cannot trace between requirements and code we cannot expect to keep up with the rate of ongoing requirements change. An Accidental Architecture, then, is a petrified architecture, and the process of hacking that underpins it is a prescription for legacy system petrification.

Known Exceptions: Perhaps in a small system, where the code is sufficiently manageable for the developer to construct a complete mental model of its current implementation, the scattered and submerged support for requirements may be not so problematic. Small systems, however, tend to grow into large systems unexpectedly, and we need to be aware of when the situation is getting out of hand. Of course, with “throwaway” systems, such as short-lived prototypes, we may be able to get away with implicit business process support, since the system will not be around long enough for requirements to change. Again, though, we have seen at EDP that today’s prototype has a nasty tendency to become tomorrow’s “alpha release” of an ultimately long-lived product. Finally, if we are absolutely convinced that system requirements are set in stone and will never change, then an Accidental Architecture clearly cannot cause us to fall behind the pace of requirements changes (although it could impact debugging). Note, however, that unwavering requirements, at least in the experience of EDP, tend to be very rare, rather requirements tend to change all the time, throughout initial development and throughout the life of a software product.

Example: At EDP, business process support tends to be reflected in code very haphazardly. Frequently, as a new requirements change has come in that upsets the current architecture a talented developer has been tasked to squeeze in a neat hack that doesn't rock the boat. This seems to be true in general across the legacy systems originally developed at EDP and also those adopted through business acquisitions. Consequently, EDP now faces the common problem that it has become extremely hard to understand both how the current requirements are already supported and how new requirements can be supported. This has led to a general reluctance at EDP to address significant requirements changes due to the overwhelming difficultly of implementing them without a high risk of unanticipated side effects "toppling the whole lot over". EDP, then, is faced with the very real problem that existing customers are now held back in terms of support for changing business processes, and hence many customers have progressively been taking their business elsewhere. Furthermore, it is becoming increasingly difficult to attract new customers, since often even their initial requirements cannot be accommodated.

An added problem for EDP is that many of EDP's products support similar business processes, yet their desired merger into a common base is significantly hampered by our inability "to see the wood for the trees". The difficulty at EDP of unravelling how these multiple products track their requirements impairs their unification into a family of related products supporting shared requirements from a common base. Thus, EDP is forced to maintain multiple completely separate products, each with different development groups addressing problems of there own, rather than working together and productively on a common cause.

A.2. Ball and Chain

Name: Ball and Chain

Type: Petrifying

Summary: Tying an application tightly to a specific platform reduces portability, thus reducing customer choice, impacting market size, and potentially impairing application life-span as new platforms emerge.

Antidote: Virtual Platform (p.272)

Symptoms: A platform is a third-party interface to which a commitment has been made, and on top of which an application has been written. When customers request an application on a certain platform, and developers are unable to fulfil that request readily, the application may well exhibit Ball and Chain. Ball and Chain diminishes portability. It imprisons an application (and this is often a life sentence), reducing customer choice in terms of the platforms upon which they may deploy that application. This is particularly frustrating when a customer has already committed to a different platform (possibly for other applications), and often results in lost sales opportunities. An application may require commitment to multiple platforms simultaneously, thus compounding the problem. For example, an application may simultaneously be tied to a specific DBMS platform, a specific Networking platform, and a specific OS platform, mandating customer acceptance of all of them. The problem may appear less severe in the short term, when particularly popular platforms have been committed to. For example, commitment to Windows NT and Oracle at the time of writing will appeal to a reasonably large percentage of customers, even though an even larger percentage will remain excluded. However, platform popularity dwindles over time and over-commitment to a popular platform today may well exclude an application from portability to tomorrow's rising stars.

Causes: Ball and Chain occurs when an application has been tied to a specific and limited set of platforms. An application is tied to a specific platform if the application is reliant upon services specific to that platform alone, or if calls to the platform's interface are so intertwined with the application body that migration to another platform would be a mammoth task. An example of an extreme case of this is Vendor Lock-In [Brown et al. 1998], wherein the required platform is at the whim of a single supplier.

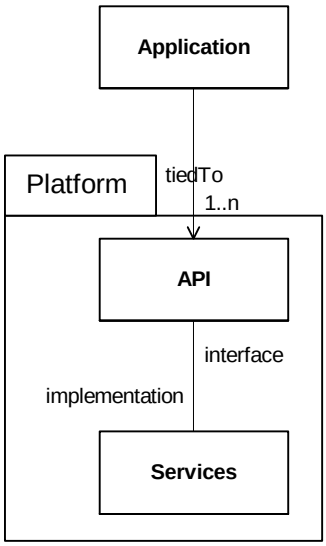

Figure A.2.1 – Ball and Chain

Impact on Legacy Systems: Tying an application to a specific platform impedes the adaptability of that application in the light of changing customer needs. In particular, if the customer shifts their commitment wholesale to a new platform or new requirements emerge which cannot be supported via the current platform, a system tied to that current platform immediately becomes a legacy system. This binding to a platform unsuitable for the reflection of evolving user requirements underlies the common perception of legacy systems as residing on old and unpopular technology, dwelling in the past, and being incapable of shifting into the modern world.

Known Exceptions: There are times when commitment to a specific platform may make sense. One example is where the requirements of the application mandate features specific to that platform. For example, a requirement that the application supports Microsoft OLE (for interoperability with other products) is a commitment in the requirements to Microsoft Windows. It is important, however, to recognise that requirements do change over time, and that what appears to be a mandatory requirement now may well change in future. Another example where commitment to a specific platform makes sense is where strict performance requirements cannot be met without that commitment. This is often reflected in commitments to specific languages (e.g. a specific assembler), to specific hardware, or to specific low level services whose performance characteristics are well known and possibly even guaranteed.

Example: Historically, EDP would sell hardware, an operating system, and application software as a complete system. Over time, customers have increasingly demanded supply of just application software products from EDP, and insisted that the application runs on an operating system and on hardware that the customer purchased from some

third party. In more than one case, such portability has proven close to impossible, and thus EDP has completely abandoned several long-standing products due to portability concerns. In order to satisfy current customer demands, and to alleviate risk as new platforms emerge, EDP has needed to substantially increase the portability of their remaining applications.

A.3. Black Hole

Name: Black Hole

Type: Petrifying

Summary: A Black Hole is a project that drags on forever. It sucks people in with no hope of escape - a dead end with no way out - and cannot attract and maintain a motivated and productive workforce.

Antidote: Negotiated Goal Alignment (p.260)

Symptoms: If a project seems to drag on forever, and developers dread being assigned to it, that project may well be a Black Hole. Other symptoms of Black Holes are: they are often pushed onto the back burner, they suffer from low staff self-esteem, there is a general lack of energy and pace necessary to tackle them effectively, and they find it hard to recruit and retain good people. Such projects are also often large, expensive, long-term, and important to the core business. Unfortunately, glamour, drive, and reward frequently shift to “sexier” novel, small-scale, and non-core-business projects. This can have a real and detrimental impact on the bottom line. Unless Black Hole projects can be made more appealing there is a real danger that they will remain resource-hungry treadmills that go nowhere.

Causes: Black Hole projects are in a rut. They persist in large part because they encourage a culture of production (wheel spinning) rather than productivity (satisfying requirements). Also, project team members tend to have been there for a long time, know the intricacies of the system inside out, and wield considerable influence over the way the project runs. The project has become heavily dependent upon them, yet these are also often the very people who perpetuate the project’s problems. Compounding the issue, there is a commonplace fear amongst potential new staff that being sucked into a Black Hole project is a dead end. Contributing to this fear is the belief that the technologies and practices often deployed in Black Hole projects are by now out of date and a "black mark on the curriculum vitae". New blood, then, is rarely attracted to the project and when it is, those already on the project often resent the newcomers, and thus strive to undermine their influence. Only those who resist change, or those hoping to hide out in semi-retirement, would choose a Black Hole as a refuge.

Impact on Legacy Systems: Legacy system projects are often Black Hole projects - we have repeatedly heard from developers at EDP that they are unwilling to get sucked into maintenance of certain legacy systems for fear that they will never get out again.

Legacy system petrification (its resistance to reflecting requirements changes), if allowed to perpetuate, can lead to developers feeling that they are on an endless treadmill, since they never seem to be getting anywhere near satisfying the ever-growing list of backlogged functionality. Furthermore, their fear (human petrification) of unintended side effects from changing accidental architectures makes attempts at changing the system deeply unsettling. The resistance of those with fresh skills to join legacy system projects and the frequently low morale and meandering focus of staff already on those projects can have a devastating effect on productivity. The consequent lack of drive on many legacy system Black Holes reduces the likelihood that there will ever be sufficient energy and pace to tackle petrification effectively, and increases the likelihood that they will remain forever "legacy".

Known Exceptions: We know of no cases where a Black Hole is desirable, with the ludicrous exception of a deliberate distraction to keep troublesome staff out of harm's way.

Example: There are a couple of Black Hole projects at EDP. People "stuck" on those projects seem desperate to escape. We have even heard one developer described as "institutionalised": imprisoned on a project, but unable to function outside its confines. These projects are going nowhere; they never seem to progress beyond "fire fighting". It is widely recognised that "new blood" is needed to give the projects sufficient energy and a fresh perspective with which to tackle them. Unfortunately, nobody with the necessary background will step forward. We have witnessed (not just at EDP) a perverse tendency to reward company *dis*loyalty, where those immersing themselves in speculative and "exciting" projects are cherished and pampered as "good guys, who really live on the leading edge". Such people are also seen to be highly marketable, and are therefore well compensated to prevent them from taking their valuable skills elsewhere. Whereas those working on less exciting applications - even those core to the business - are relegated to the status of "doers" rather than "thinkers". We have been told on several occasions that the Black Hole projects at EDP are a dumping ground for "deadwood" - developers perceived as past their sell-by date. Even where the systems stemming from such projects form the bedrock of the business and generate more revenue than any other project, we have witnessed an occasional perception that the project staff themselves are to blame for "holding back the business. If those guys had got it right in the first place, we wouldn't be in this mess." With sentiments like these, it’s no wonder people shy away from such work; most people want to be associated with success, not failure.

A.4. Bound and Gagged

Name: Bound and Gagged

Type: Petrifying

Summary: Hard-coding available object types in an application can lead to increased application downtime, application bloat, higher costs, and lower requirements turnaround time.

Antidote: Dynamic Pluggable Factory (p.229)

Symptoms: When there is significant and frequent application downtime to support the inclusion of new object types, the application is probably Bound and Gagged. That application downtime impedes business process support for its duration, and will be increasingly unacceptable as user expectation shifts to twenty-four hour per-day service time (e.g. e-commerce). A second symptom is where the application cannot be adapted to its local context without either providing source code or going back to the original developers for a new version of the executable. Handing out source code may prove unacceptable for both product support and intellectual property reasons. Going back to the original developers can reduce turnaround time, increase costs, and lead to application bloat.

Causes: When the types of object that an application can create are hard-coded (bound) in the body of the application, that application is then unable to talk about objects of other types (i.e. it is gagged). When new types of objects are required, the only option is to shut down the application, and install a new version of the application with the new types of objects added.

Impact on Legacy Systems: Hard coding all object types into an application both potentially complicates the internals of that application, and mandates that all requests for new types of object must be accommodated via yet further complication of the internals of the application. The increased complexity of the application reduces maintainability of the application, encouraging architectural petrification. Furthermore, the increased workload on maintainers of the legacy system to accommodate new object types simply adds to the growing backlog of requirements from which the legacy system is slipping further and further away.

Known Exceptions: For many applications, although interactions between objects may need to change in response to new requirements, the actual types of object required is well known in advance and is relatively stable. In many shrink-wrapped applications,

customers do not have varying requirements with respect to required object types, and hence the application is not Bound and Gagged.

Example: A recent Internet-Browser-based application, developed at EDP, originally hard-coded the types of dynamic visual object (WebWidgets) that could be incorporated in a web site. At first this wasn't a problem, since the required number of WebWidget types was small and manageable. Unfortunately this requirement grew unexpectedly quickly. As a consequence, the body of the application became rather complicated, since it embedded knowledge of, and made assumptions about, each of the growing number of WebWidget types. Adding support for additional types became a major challenge since only one person understood the intricate code well enough to be trusted with its modification (which made the code yet more intricate). All other developers then had to wait for that single developer to incorporate new WebWidget types into the application. In addition, the hope that, in the future, third parties would be able to provide additional types was thwarted, since they would not have access to the application source code. It became essential, then, to eliminate the dependence of the Internet-Browser-based application upon specific hard-coded WebWidget types.

A.5. Code Pollution

Title: Code Pollution

Type: Petrifying

Summary: Problem-domain-oriented code tends to be wrapped in guard-code that enforces implicit time-ordering dependencies throughout the system. This guard-code pollutes the code space, and hinders maintainability, adaptability, understandability, and reusability.

Antidote: Explicit Protocol Reflection (p.235)

Symptoms: When the "real" (problem-domain-oriented) functional code within a system seems to be buried under masses of conditional code and state manipulation code, which appear unrelated directly to the business value that functional code delivers, the system may be suffering from Code Pollution. An additional symptom is where developers find it difficult to unravel and make explicit the legitimate time ordering of function calls because the only recording of this ordering is within the scattered code that enforces it. Thus it becomes difficult for the maintainers of a component to comprehend and maintain its time-ordered collaboration protocols, and it becomes equally difficult for users of such a component to learn and respect such constraints. As a component becomes more sophisticated, with the addition of new data members and methods, and new time-ordered collaboration protocols, Code Pollution becomes more and more problematic. The increasing need to reflect growing intra-component dependencies across methods and data members leads to fragile components unable to reflect further business process change due to the fear that "if we touch it, it might break".

Causes: The function signatures in a component's interface effectively detail the events in which the component is interested. By default, the order in which those events may be received by the component is completely unconstrained. This unconstrained ordering is often problematic. For example, it would be inappropriate to allow an update lock to be set on a database record that was already locked. Thus, the events that a component is interested in receiving varies from moment to moment as a function of the mutations to its abstract state which have occurred in response to the events that it has already received. That is, the invocation ordering of a component is historically determined and hence mutable. A given component, then, exhibits an implicit time-ordered protocol. To manage time-ordered dependencies, and enforce respect for time-ordered invocation, it

is commonplace to introduce both state-tracking variables and guard code that manipulates and interrogates those variables. This code is then wrapped around the functional code that actually delivers business value. Over time, as changing requirements encourage system growth, the lattice of state-tracking variables and guard code becomes more and more convoluted, obscuring the actual functional code, and challenging the ability of developers to comprehend and maintain the behaviour of the system.

As an example of Code Pollution, taken from [Lauder and Kent, 1999a], imagine a component that represents an application for a loan, as shown in Figure A.5.1. For the sake of simplicity, only a minimal set of attributes and methods are depicted.

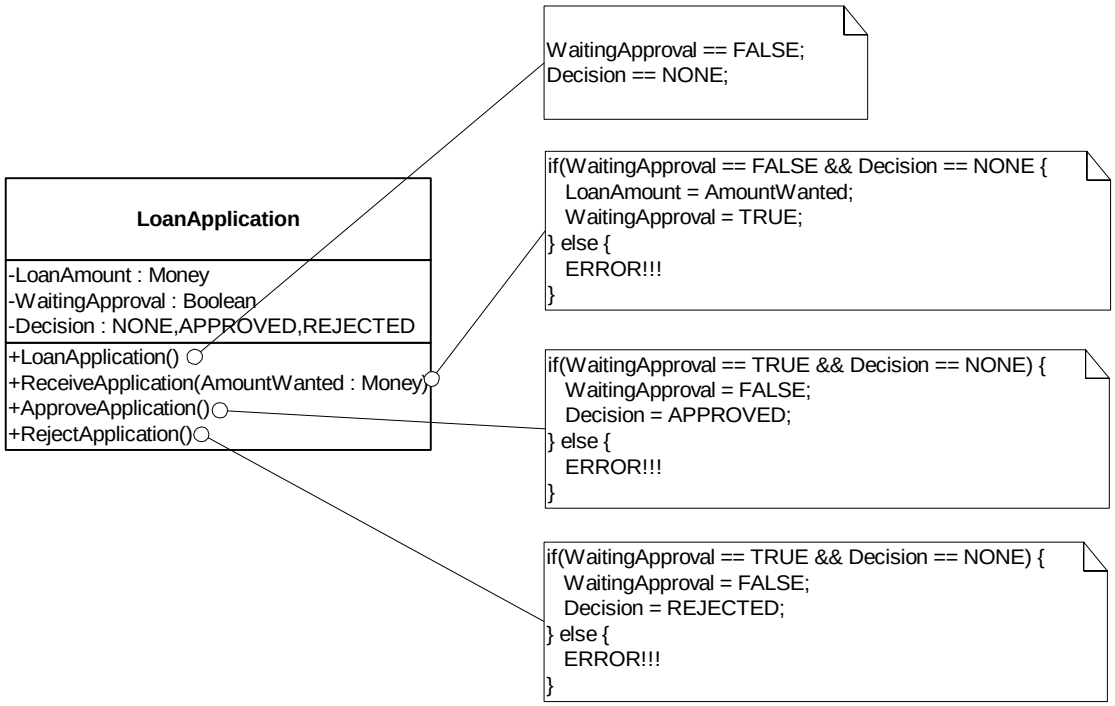

Figure A.5.1 - Code Pollution

In Figure A.5.1, a LoanApplication at first represents, in effect, a blank piece of paper. When an actual loan application is received, the requested loan amount is recorded. The requested loan may then be either approved or rejected. Notice that each method is surrounded by conditional code and contains a variety of flags being set to control and indicate progress of the loan through an implicit ordering of method invocations. The implicit protocol is that a LoanApplication must be blank when an actual loan application is received, and a loan can only be approved or rejected if that loan has been received and has not already been approved or rejected. The individual methods in the LoanApplication component, then, have become polluted with code which is not specific to the fundamental behaviour of those methods, but which is essential in case

methods are invoked out of order. As the LoanApplication component becomes more sophisticated, with the addition of new data members and methods, we can expect Code Pollution to become more and more problematic.

Impact on Legacy Systems: Legacy system petrification stems in large part from implicit architectures layered on top of a system's legitimate architecture (see Accidental Architecture, p.178). Implicit architectures are "hacked" into systems wherein the developer is struggling to understand the purpose of existing code and is wary of the potential impact of changes they wish to make. Code Pollution encourages such hacking, since the complex lattice of dependencies spread throughout the code leads to a fragile architectural base. Here, changes in one area can ripple across to another part of the system by upsetting the intricate web of dependencies that both of those parts of the system must respect. The resultant fear of doing harm leads developers to localised hacking in an attempt to stem potential side effects resulting from more global code changes. Each hack effectively constructs another layer of implicit architecture, as localised guard code is further complicated to compensate for state dependency concerns which were not well understood. This, of course, compounds the complexity of the architecture, reducing comprehension further still, deepening system petrification - making the legacy system yet more change resistant.

Known Exceptions: Code Pollution is a matter of degree. There are many applications where ordering dependencies are minimal. In such circumstances guard code may not swamp functional code particularly harshly, and thus Code Pollution may not be particularly damaging (with respect to system comprehension and maintainability). Nevertheless, developers must be vigilant against the stealthy creep of further Code Pollution as system evolution introduces more and more guard code, and as their own understanding of the intricacies and intertwining of this code diminishes.

Example: All systems examined at EDP suffer, at varying levels of severity, from complicated lattices of guard code. Developer expertise at EDP, with respect to the source code for a given legacy system, seems to relate not so much to comprehension of the behaviour of individual functions but rather to an appreciation of the subtle relationships that hold between multiple functions. The behaviour of individual functions is generally relatively easily determined by localised code inspection. Absorbing dependency lattices, on the other hand, appears to take lengthy and deep immersion in the guts of the system. This causes EDP significant problems, since only developers who are extremely familiar with such dependencies can be trusted to make

worthwhile changes to a system. Neophytes are essentially rendered impotent. This has led to a culture of long-timers over-burdened with endless backlogs of change requests, and intimidated newcomers forced to thumb-twiddle for fear of their changes back-firing. There are, unfortunately, parts of many of EDP's legacy systems where even the most experienced developers fear to tread. Comments stating "not sure what this does, but don't touch it just in case" are not uncommon. This state of affairs is, of course, unacceptable both to the developers themselves, and to business people and customers who fail to receive adequate support for the business-value laden requirements changes they need.

A.6. Gold In Them Thar Hills

Name: Gold In Them Thar Hills

Type: Petrifying

Summary: System requirements are often lost in time, and scattered throughout many sources, each with only a partial (and often blurred) understanding of the business processes the system is intended to support. Consequently, as we evolve the system in response to changing requirements there is no clear basis upon which to determine an appropriate architecture. This leaves us effectively stumbling in the dark hoping things will "work out in the end".

Antidote: Gold Mining (p.242)

Symptoms: When we are unsure of the requirements the system is currently satisfying, and can only describe them primarily in technical terms rather than business terms, our understanding of the business processes underpinning those requirements has probably been lost in history. When a business-oriented understanding of requirements seems lost we may well be suffering from Gold In Them Thar Hills.

Causes: To maintain a legitimate architecture (see the Legitimate Architecture productive pattern, p.254) for an information system, we need to continually evolve our understanding (our "mental model") of the changing business processes that the information system is intended to support. Frequently, the original developers of a mature information system have moved on, leaving behind newcomers with only a partial understanding of both how the system hangs together and where its requirements stem from. Indeed, it has been said by numerous authors (including ourselves, see for example an early version of *Gold Mining* in [Lauder and Kent, 2000b]) that an information system is frequently the only reliable repository of supported-business-process expertise. However, we have now realised that this is usually an oversimplification. More realistically, it seems, at least from the EDP experience, that an information system is often an almost accidental artefact resulting from an amalgamation of a series of partial understandings of its evolving requirements. Thus, initial requirements, and each subsequent requirements change, have been realised in code with only a minimally sufficient grasp of the overall requirements of the system. Overall business process support then becomes a patchwork of support for this progression of requirements fragments. Importantly, the edges between these fragments are often very rough, and the stitching between them tends to be untidy. At these edges

we will find buried support for old requirements that are no longer needed, and hacked in support for unanticipated requirements that have been force fitted into an architecture that was not designed to accommodate them. That is, the system is just about hanging together. If we appeal to what has been termed the Jedi principle ("use the source Luke") and interrogate the legacy system as a source for understand its requirements we will uncover only the amalgamation of this patchwork of partial original requirements understanding.

Nor it seems can we turn primarily to individual "domain experts" (as we naively thought in our earlier work [Lauder and Kent, 2000b; Lauder and Lind, 1999]), since we have found at EDP that domain experts are often a mythical metaphor for (legitimate) "buck passing" rather than existent oracles of domain expertise ("I implemented it that way because the business people told me to"). This is fine for pinning down requirements for new features (indeed, it underpins the requirement gathering strategy of Productive Migration and Productive Evolution). We cannot, however, pin our hopes mainly on a few "domain experts" to recover lost (tacit) knowledge of requirements that have already been implemented. In reality, just as domain knowledge tends to be scattered throughout code, we have seen at EDP that domain expertise tends to be scattered throughout the heads of many different people – business people, developers, external authorities, and so on. Thus, there is rarely a single point of reference for domain understanding.

The problem, then, is that we know the implicit requirements are out their somewhere (hence, there's Gold In Them Thar Hills), but attempting to gather all the revelant knowledge sources together, filter through them, and reconstruct legitimate system requirements as a basis upon which to discuss and accommodate future requirements change is a potentially overwhelming task. There is a tendency, then, at least in the experience of EDP, to "fumble around in fog", unaware quite where the system has come from and thus never fully understanding where it is going next, hoping that future requirements changes will "sort themselves out in the long run" in the words of one EDP developer.

Impact on Legacy Systems: If we don't really have a good overall understanding of what the cumulative system requirements are we can't expect to have a legitimate architecture grounded in the commonality and variability needs of the business processes being supported. As a consequence, the system's architecture becomes an accidental consequence of multiple waves of hacking to force fit our (partial)

understanding of an immediate requirements change into a base whose requirements we do not actually understand. The eventual result is both architectural petrification, wherein it becomes close to impossible to absorb new requirements change, and human petrification since our “fumbling in the fog” is an unnerving experience wherein we are afraid of making further changes.

Known Exceptions: It is hard to think of a situation where the loss of requirements understanding is not problematic. Perhaps for the very rare circumstance where a system is set in stone and will never need to change then the loss of knowledge will not be a problem.

Example: It is interesting to listen to developers at EDP talk about user interaction with the company’s information systems. A typical narrative goes like this: “They run CS234 before doing a PS15, then once a month they run off a RP983”. To an outsider it is close to impossible to understand such conversations. Essentially, developers have cultivated a vocabulary stemming from the system implementation (specifically, the names of subroutines). This may serve as useful shorthand for the developers themselves, but it also underlies a problem that such terms now underpin the primary understanding of the business domain for many people at EDP. Knowledge of the actual business processes being supported, as seen from a business viewpoint, has been lost over time, so that only fragments remain in the mental models of a few “old-timers”. It is extremely difficult, then, for EDP developers to map business process change to corresponding code changes, since that business process change must first be mapped into internalised jargon such as “CS234”, and so on. As “old-timers” have moved on or even retired from the company, the expertise with which to perform this mapping has diminished considerably. Historically, there has been little progress at EDP in terms of recapturing lost business knowledge – widely scattered throughout many sources – so that the current wave of developers are unable to cultivate satisfactory mental models with which to talk about, explore, rationalise, and elaborate system requirements in business terms.

A.7. Human Shield

Name: Human Shield

Type: Petrifying

Summary: Many applications were developed with the assumption that only trained and trusted employees would interact with them. Letting customers interact directly with a company's computer system exposes them to the inadequacies of the computer system, exposes the computer system to the customers' inexperience and possible malicious intent, and effectively delegates the external face of the company to a chunk of inanimate software.

Antidote: Firewalling (p.239)

Symptoms: When customers are encouraged to directly interact with a company's applications, and those customers find that interaction confusing and intimidating, or where the applications are being misused maliciously by outsiders, those applications were probably designed with a Human Shield in mind.

Causes: Business processes often involve interaction with customers. The software supporting such business processes is usually intended only for trained experts at the organisation enacting the supported business processes (see Figure A.7.1).

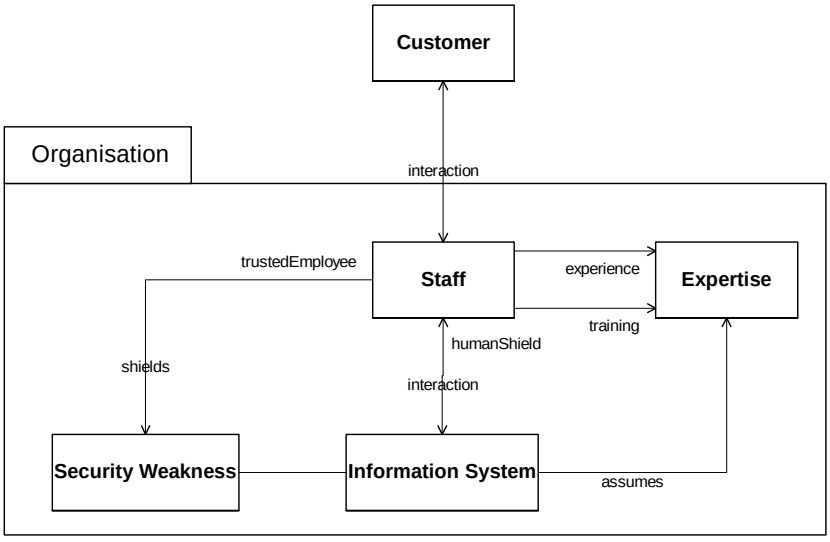

Figure A.7.1 – Human Shield

Such experts effectively form a Human Shield between the customer and the computer system. The presence of a Human Shield has two implications. Firstly, we can assume that the user of the computer system has undergone appropriate training and has access to suitable documentation, making them expert in its operation and experienced in

translating a diverse range of customer needs and requests into appropriate sequences of computer system interaction. The second implication is that the primary interface to the company is a human being, acting as a multi-talented, interactive, warm-blooded buffer that hides any internal inadequacies of the computer system from the customer.

Despite the many inherent advantages of a human face to a company there has been a noticeable shift in demand to bypass the Human Shield and allow direct customer interaction with a company's computer software. One direction of the shift has been towards load-and-go applications (often automatically installed from a CD) given to, say, a purchasing customer by a company that sells goods. Usually, there is minimal documentation supplied with the CD and there can be no presumption that relevant training has been undertaken or is even available. Typically, the CD contains the seller's catalogue of goods for sale and an order entry application via which orders are placed and then transferred electronically to the seller's computer system. The purchaser's normal interaction with the seller, then, has shifting away from a human being towards the software supplied. Explosive increase in the demand for Internet-based e-commerce adds the additional dimension that we can no longer be sure even who the user of the information system will be. Here, the seller's catalogue is maintained on a web site and is (often) made available to anybody with an Internet connection.

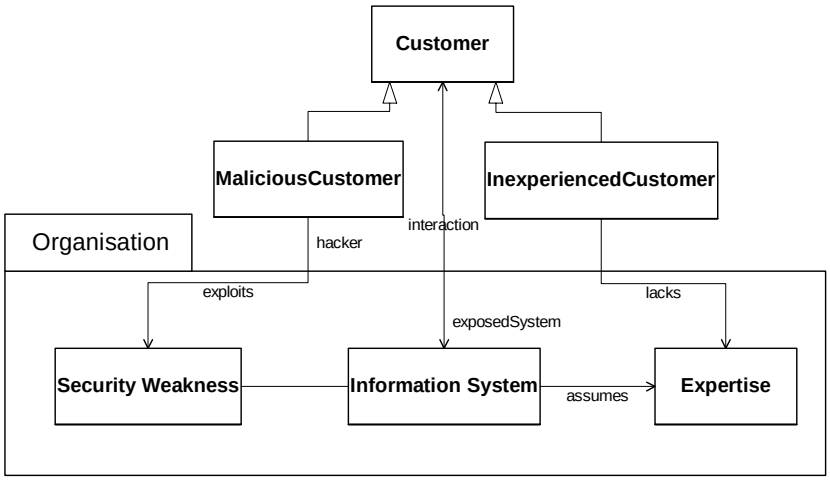

Figure A.7.2 – Loss of Human Shield

When information systems were not designed with consideration for external users, those information systems will support the business processes of our own organisation, rather than the business processes of the customer. This forces the customer to work in ways that suit us, rather than ways that suit them. Consequently, they see our information systems, and the business processes it supports, "warts and all". This

exposes the user to the inadequacies of our information systems, and exposes our information systems to the inadequacies (and possibly malicious intent) of outsiders.

Impact on Legacy Systems: Legacy information systems were usually developed with a Human Shield in mind. The underlying assumption that only trained and trusted users will access them is no longer valid. Legacy systems resists ongoing business process change, and hence their inadequacies in supporting current business processes are often masked by humans filling in the gaps. The loss of the Human Shield and the absence of training, documentation, and access to experienced colleagues for external users, mean that we cannot protect customers from the complexities and the functional inadequacies of the legacy system anymore. In addition, legacy systems often employ simplistic security mechanisms, since they were usually developed without concern for remote user access, and updating their security is challenging due to their inherent change resistance. The legacy system, then, is exposed to mischievous or malicious attacks. Since determined hackers no longer have to go though a Human Shield to reach the computer system, we must be protected against them in other ways. Finally, when a customer's primary interaction with an organisation is via its computer system, the customer's perception of that company, and the reputation the company is left with, is formed from the customer's impression of the efficacy of the computer system in terms of satisfying that customer's particular concerns. In effect, the company has deliberately placed its reputation with an inanimate chunk of legacy software, with none of the people-skills traditionally associated with establishing and maintaining good customer relations.

Known Exceptions: EDP has discovered that many customers actually prefer a Human Shield, even where they could access the computer systems directly themselves. For many customers in the construction industry, for example, the social interaction inherent in the Human Shield is seen as a fundamental part of a sales transaction. We have seen customers who deliberately travel to a sales counter each morning, or make a phone call instead of using the Internet to place an order. These customers prefer human rather than computer interaction, it transpires, in order to exchange gossip about competitors, keep their finger on the pulse of trends in their industry, and establish long-term relationships wherein give and take is the norm. In these circumstances, then, the Human Shield is an asset, not a liability.

Example: EDP develops software for organisations that sell and distribute goods. That software is, for the most part, designed with trained expert users in mind. Historically,

when a customer places an order - typically via the telephone or at a sales counter - a trained employee of the selling company enters that order into the computer system on the customer's behalf. The information system interface assumes that the user has a high level of knowledge of the product being sold: for example, obscure product codes are often entered directly, and efficient "magic keystrokes" navigate the user around the system rapidly. On the telephone or at the sales counter, time is money, and the information systems sacrifice bells and whistles in order to accommodate expert system interaction. Opening up these systems to the outside causes EDP two problems: firstly, the user interface is far too complex for inexperienced users, and secondly, the systems are far too "open to attack" to risk malicious outside access. Attempts by EDP to accommodate secure Remote Order Entry have proven quite cumbersome and thus only suitable for EDP's own highly trained employees ("roving salespeople"), whereas external customers have always ultimately needed to rely on a Human Shield being in place.

A.8. Legacy Customers

Name: Legacy Customers

Type: Petrifying

Summary: Dependency upon customers who resist change can prevent us from tackling petrification, moving a software system forward, and attracting new customers.

Antidote: A Spoonful of Sugar (p.220)

Symptoms: Legacy Customers are generally indicated when: (1) Long established customers make demands that are contrary to the needs of newer customers. (2) The demands of those long established customers invariably take precedence. (3) Newer customers are lost due to the lack of responsiveness of the software supplier.

Causes: Legacy Customers are generally those who are happy with the system in its current form, do not wish it to change in any way that would affect backward compatibility, and significantly resisting such change. As an example, green-screen adherents (see the Ugly Duckling petrifying pattern, p.214) are frequently Legacy Customers. Usually, Legacy Customers have been an invaluable source of recurring revenue for the software supplier. They are often customers with whom the software supplier has a well-established long-term relationship, strengthened through years of partnership. They have frequently established strong personal bonds with individuals at the software supplier, some of which become strong supporters of their cause. We have found that, often, there are direct links to (now) upper management, whereas the wishes of newer customers frequently must filter through lower level sales and support staff. All in all, Legacy Customers are generally valued highly for their continued loyalty, and companies don't want to let them down. All of these forces lead to a considerable degree of influence in the hands of the Legacy Customer. Unfortunately, they often wield this influence in directions that are contrary to the needs of newer customers, stifling new revenue, and thus can hold back both a legacy system and the company supplying it.

Impact on Legacy Systems: Legacy Customers perpetuate the status quo. They discourage the maintainers of a legacy system from moving it forward. It becomes extremely difficult to tackle petrification without "rocking the boat" in the eyes of Legacy Customers. For example, applying Lipstick on the Pig (to inject a graphical user interface, see p.257) is unlikely to be popular where Legacy Customers have made a significant financial investment in dumb terminals, and whose employees are highly experienced with using the system as it currently stands. Furthermore, since Legacy

Customers are substantially satisfied with the current offering, they are generally unwilling to spend extra money on additional features, thus starving the software supplier of sufficient funds with which to develop new features to entice new customers. Financial constraints, and dependency upon recurring license or maintenance fees from Legacy Customers, can force the software supplier to focus primarily towards supporting the current system rather than striving to move it forward. That is, remaining loyal to Legacy Customers is often in conflict with tackling petrification, and hence can perpetuate a petrified legacy system.

Known Exceptions: Paradoxically, Legacy Customers are potentially a dream come true. If their demands are solely to maintain the product unchanged, and they are willing to pay an ongoing fee for us to do so, then they are almost a license to print money. We want more customers like this, not fewer. Here, we can simply freeze a product, provide a minimal support service for it, and let continuing revenue keep rolling in. Legacy Customers, then, are only a problem where their demands for stability interfere with the demands of (typically) newer customers. If there is no such conflict, and they are happy with an essentially frozen product, we can "keep milking the cow" for as long possible.

Example: EDP has had numerous Legacy Customers, many acquired through business acquisitions. This is particularly troublesome, since EDP's general direction is to merge the software assets of its various subsidiaries into a single unified software suite. Customers who strongly resist such change have given EDP considerable challenges in terms of both advancing product offerings whilst also maintaining backwards compatibility. Eventually, the tensions have reached crisis point, at which stage EDP management has been forced to make difficult decisions with respect to how to tackle the Legacy Customers that threaten to hold the business back.

A.9. Monolithicity

Name: Monolithicity

Type: Petrifying

Summary: When a system is constructed as one or more indivisible monolithic large-scale "chunks", understandability, reliability, and reusability suffer.

Antidote: Virtual Componentisation (p.269)

Symptoms: When developers become swamped in the mass of details of a system's implementation, and "can't see the wood for the trees", it may be because the system is essentially monolithic. This reduces their comprehension of the system's internals, and thus the reliability of any changes they make, since it is effectively necessary to understand everything in order to fully understand anything. Furthermore, opportunities for reuse or substitution across the monolith are minimised, since developers are unable to isolate reusable parts from the rest of the system. Another symptom is "bloatware", wherein the system becomes a dumping ground for all features requested by all users, encumbering each user with the aggregate of features requested by everybody (see also Swiss Army Knife [Brown et al. 1998], and Gilding the Lily [Brown et al. 2000]).

Causes: Monoliths are often simply too big to manage. Bizarrely, though, we have noticed considerable pride in some quarters about the immensity of their monolithic systems: "five million lines of code in there!", and a delight in how the systems grew from small beginnings into what they are today. The sentiment seems to be "great oaks from little acorns grow". This may be true for trees, but it is not true for software. At least not without substantial nurturing, pruning, and reshaping along the way.

Monolithicity often results from a system that grew organically (a “Big Ball of Mud” [Foote and Yoder, 2000]); rather than resulting from some up-front design or ongoing re-evaluation of architecture. Thus, the system grows bit by bit, with more being added almost haphazardly to the monolith as new requirements emerge. The result is often Spaghetti Code [Brown et al. 1998], littered with no-go areas (such as Lava Flow [Brown et al. 1998] code). Unravelling the intricacies that proliferate throughout the code often requires an encyclopaedic knowledge of the development history of the system, in order to understand why things are done a certain way, and what assumptions underlie the current implementation (see also Gold In Them Thar Hills, p.192). Of course, when only a small band of old-timers can work on the system, their practices are

likely to proliferate, and it is unlikely there will be either incentive or experience to improve the situation (see also Black Hole, p.184).

Impact on Legacy Systems: One early advantage of a monolith is often that conceptually distinct parts of the system are seamlessly integrated (after all, they are all part of the same code). Such seamlessness is often harder to achieve with parts of a suite that have been developed independently (the seams almost always show, no matter how neat the stitches). The seamlessness advantages of monoliths, however, are often quickly eroded as the immensity of comprehending the whole of the system overwhelms the abilities of developers, who then resort to hacking. Here, implicit architectures are layered upon top of the monolith, in an attempt to tame the beast and localise the scope of change. The resultant lattice of implicit architectures creates a complex tower balanced on top of an unstable base (see Accidental Architecture, p.178). The result, of course, is progressive petrification, so that eventually, the system has essentially solidified against any significant future requirements change.

Known Exceptions: Monolithicity is probably a matter of scale: both of code size, and change frequency. If the system is relatively small (maybe below a few thousand lines of code) it is perfectly feasible for developers to maintain a comprehensive mental model of the system internals in their heads. With a good mental model, developers can usually adapt the system quite radically, and thus Monolithicity is unlikely to be a major problem. On a larger scale, a system spanning hundreds of thousands of lines whose requirements almost never change is probably not worth tackling: "let sleeping dogs lie". At the very largest scale (millions of lines of code), the only possible case where Monolithicity would seem to be acceptable is because we are going to let the system die. This appears to be a common strategy, where a product is eventually discontinued because the economics of prolonging its life don't add up.

Example: A number of applications at EDP are monoliths. The main distribution management is an excellent example of a product that has simply grown and grown over many years into the gigantic monolith that it is today. Few developers have sufficient immersion in the history of this project to understand the convoluted design assumptions littered throughout its internals, and fewer still have the drive to tackle the monolith head on. Due to the monolithic nature of this application, it has not been possible to provide component substitution where customers demand, for example, a different stock control system, or a different financials package. The customer must “take it or leave it” as it is, and increasingly customers are choosing to “leave it”.

A.10. Persistent Problems

Name: Persistent Problems

Type: Petrifying

Summary: Intertwining of intricate rules for persistence with business-process-oriented application logic obscures both concerns, thus impeding the understandability of the application and reducing its maintainability.

Antidote: It's Not My Concern (p.251)

Symptoms: When code interacting with persistent stores such as databases and filing systems detracts from the flow of the application, thus impairing its understandability and maintainability, the system is probably suffering from Persistent Problems.

Causes: A significant part of an information system's implementation code typically involves reading from and writing to a persistent store, such as a database or a filing system, and managing interdependencies between various fragments of related data across that persistent store. This interaction with persistent storage tends to be intermingled with and clutters business application logic. Often the rules of persistent storage interaction are rather convoluted, involving multiple interrelated reads and writes across multiple associated files under the control of dependency rules embedded in the application code.

In essence, this intertwining is a deleterious tangling of two separate concerns, introducing dependencies between them that are an accident of implementation rather than inherent in their relationship. Tangling of separate concerns complicates application logic, making it difficult to trace implementation decisions back to the original application requirements and to trace ongoing changes in those requirements forward to appropriate changes in the application

Impact on Legacy Systems: The complexity of code and lack of traceability between requirements and implementation that can result from Persistent Problems are major impediments to the reflection of ongoing requirements change, and hence help perpetuate the backlog of unsupported requirements changes in a legacy system.

Known Exceptions: The problem may, of course, be that the application logic is simply unmaintainable, and the persistence code is the "fall guy". Certainly, it is appropriate to refactor the code in question before blaming interactions with the database.

Example: EDP included, in the body of a large order processing application, logic to track and log changes to order-related records. This was a secondary requirement to the actual maintenance of the order records themselves, and hence could not be traced directly back to business process needs, but was still recognised as necessary functionality.

```
read product from productFile using productId
product.quantity -= orderLine.quantity
write product to productFile using productId
write product to productLogFile using timeStamp + productId
orderLine.cost = product.unitPrice * orderLine.quantity
write orderLine to orderLineFile using orderLineId
write orderLine to orderLineLogFile using timeStamp + orderLineId
read order from orderFile using orderId
order.orderLineIds += orderLineId
write order to orderFile using orderId
write order to orderLogFile using timeStamp + orderId
```

Figure A.10.1 – Persistent Problems

Developers had to be very careful when making changes throughout the application to code that affected order records (and other associated records), to ensure that those changes were also reflected in corresponding tracking and logging code. This proved to be an intricate, tedious, and error prone process, and the resulting code tended to be difficult to follow and hard to maintain. Furthermore, outside the application, records would occasionally be edited manually, and such changes were beyond the reach of the tracking and logging code. Thus, related files could become inconsistent with respect to one another, mandating development and periodic utilisation of utilities to reconcile these inconsistencies. There was also a need to import new orders that had been submitted electronically and stored in standard Unix files rather than entered via the normal keyboard route. These Unix files were separate from the database management system, and hence had to be imported, and their format converted as necessary, using alternate mechanisms from those used for normal orders. The converted records were ultimately stored in the orders database wherefrom they could be queried as if they were orders entered in the normal way. Note that, in order to maintain performance, various cross-references were maintained directly by the application to map from foreign keys to frequently required associated data in other tables. All of this database maintenance code was a tremendous burden on the application, increasing its complexity, and stretching to the limit the ability of application developers to comprehend and maintain that code.

A.11. Reuse Abuse

Name: Reuse Abuse

Type: Petrifying

Summary: Failure to reuse artefacts in product development increases the time and costs associated with product delivery whilst also reducing the quality of those delivered products.

Antidote: Systematic Reuse (p.266)

Symptoms: When development productivity never increases, despite the fact that developers appear to be tackling a series of related problems, the organisation may be committing Reuse Abuse. The eventual products emerging from such Reuse Abuse are likely to be of lower quality and more error prone than equivalent products constructed with fine-tuned and field-proven robust and reusable artefacts. Reuse Abuse frequently increases application size and complexity, and impedes maintainability. In the long-term, it increases software development time, since considerable effort must be spent either on reinvention, re-implementation, and debugging of new artefacts, or on struggling to force ill-fitting artefacts to work in the intended reuse context. When requirements changes occur, it then becomes necessary to reflect this across all copies of a cloned artefact, rather than in a single common base, and to rethink and re-implement force-fitted artefact reuse where that change conflicts with localised contortions. The extended delivery timeframe inherent in long-term Reuse Abuse comes at a higher financial cost, and impacts the availability of resources for other work. Furthermore, the delay in delivery means that customers must continue to work for longer without satisfactory support for their evolving business needs.

Causes: Reuse Abuse is about misusing opportunities for reusing what already exists and also about creating artefacts without reuse in mind. Reuse Abuse applies not just to program code, but across all artefacts of the software development process, including requirements, designs, tests, training material, and so on.

Impact on Legacy Systems: The increased development time resulting from long-term Reuse Abuse perpetuates the lag inherent in legacy systems between requirements change and implementation. When requirements changes are finally delivered, they tend to be at a higher cost and often of a lower standard than when appropriate reusable artefacts are used, thus perpetuating an ongoing maintenance burden. Reuse Abuse,

then, prolongs the life of a legacy system and diverts time and money away from the pressing need to actually migrate that legacy system to an adaptable replacement.

Known Exceptions: Where different teams work on divergent projects, using different technologies, and are dispersed widely geographically, it is hard for them to avoid Reuse Abuse. The effort involved in maintaining a common reuse culture across such an environment may be greater that any potential saving in time, artefact size, and quality.

Example: There are two main forms of Reuse Abuse seen in the systems studied at EDP. The first form is where an opportunity to reuse an existing artefact is not taken, the second form is where reuse actually is undertaken but it is poorly executed. Examples of lost opportunities for reuse seen at EDP include Reinventing the Wheel [Brown et al. 1998] and Short Changing. Reinventing the Wheel is where the same concept is reflected in multiple diverse implementations scattered throughout the system, without care for ensuring their commonality. Short Changing is where functionality is excluded from a system because its development would be too expensive, even though much of that functionality is already encapsulated elsewhere in reusable artefacts. Examples of poorly executed reuse include Copy-and-Paste and Square-Peg-In-A-Round-Hole. Copy-and-Paste is where an existing artefact is cloned and that clone is then pasted into its new context, wherein it is adapted and evolved independently of the original. A commonplace example of this is Cut-and-Paste Programming [Brown et al. 1998; Brown et al. 1998]. Square-Peg-In-A-Round-Hole is where an existing artefact is force fitted inappropriately into a new context and contorted at its interface to provide functionality for which it was never intended.

A.12. Tight Coupling

Name: Tight Coupling

Type: Petrifying

Summary: When conceptually distinct parts of a system are tightly coupled to one another, it makes them extremely sensitive to changes in each other's interface and implementation, which hinders maintainability, adaptability and reusability.

Antidote: Implicit Invocation (p.246)

Symptoms: When conceptually separate yet collaborating parts of a system are highly sensitive to changes to one another's interface, implementation, or even existence, they may be suffering from Tight Coupling. This dependency increases the likelihood of propagated side effects resulting from what should be localised code changes. This, in turn, increases the potential for the introduction of unanticipated and subtle bugs, and reduces the level of confidence of developers that their changes are free from harm.

Causes: Tight Coupling results when different parts of the system exploit knowledge about one another that is incidental to the behaviour they wish to invoke, and indeed essentially an accident of its implementation rather than intrinsic to the behaviour itself. In the most blatant cases, this may involve direct manipulation of one another's data members. Adherence to concepts of information hiding has, thankfully, reduced (but not eliminated) the incidence of such practices in recent years. Still commonplace, however, is the direct invocation of one another's methods via signature matching in a process termed *explicit invocation* [Shaw and Garlan, 1996]. This results in a Tight Coupling between distinct parts of the system by making them dependent upon the stability of one another's interfaces and, perhaps more subtly, of one another's existence.

Impact on Legacy Systems: Where different parts of the system are tightly coupled to one another, it hinders the adaptability of those parts due to the potential "ripple effect" of changes. For example, changing function interfaces can have a cascading knock-on effect as parts of the system that are tightly coupled to the current interfaces need to be changed accordingly. As another example, removing one part of the system and replacing it with, perhaps, two smaller replacements each taking over half of the work of the former can have a major impact upon other parts of the system, which were tightly coupled to the assumption of that larger initial piece of the system being in place. These examples are commonplace in the legacy systems we have examined and they significantly increase the likelihood of system petrification by making the system more

sensitive to change, thus making developers less willing to make such change, lest unexpected side effects result. The consequence of this human petrification is, of course, that a backlog of pending changes builds up so that the system becomes progressively further misaligned with the business needs it is meant to support.

Known Exceptions: In some circumstances, Tight Coupling actually makes sense. When pieces of a system are actually closely conceptually related, and can be viewed together effectively as a larger grained black-box component by the rest of the system, then explicit invocation is probably harmless. Indeed, adhering to Tight Coupling in such circumstances may increase the developer's immediate comprehension of how these closely knit components relate to one another, reducing human petrification. Of course, this is a question of balance: even in cases of high cohesion, we need to guard against excessive exploitation of respective implementation knowledge due to the consequent reduction in maintainability it can induce.

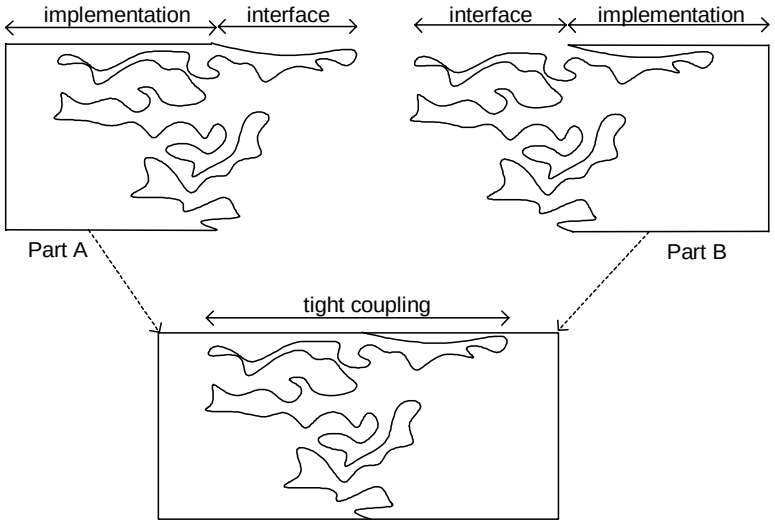

Figure A.12.1 – Tight Coupling

Example: EDP's sales order processing system is deeply intertwined with their own stock control system. Scattered throughout the sales order processing system are extensive explicit invocations of intricate low-level signature-bound functions of the stock control system. This deep dependency of the sales order processing system upon the specific interface of the stock control system has ensured that attempts to satisfy repeated customer requests to substitute other stock control systems have been wholly unsuccessful.

A.13. Tower of Babel

Name: Tower of Babel

Type: Petrifying

Summary: Developing applications in different programming languages often impairs interoperability and substitutability thus undermining customer demands for cohesive application suites.

Antidote: Babel Fish (p.222)

Symptoms: Customers increasingly demand that applications should be able to inter-operate as a suite. They also anticipate the ability to substitute different applications or parts of applications in place of those provided in the suite. Where the diverse implementation languages used in those applications impede such interoperability and substitutability, the system is exhibiting the Tower of Babel pattern (named after the biblical story of the divisive nature of separate languages).

Causes: Programming languages tend to manifest assumptions about the implementation technology of components with which they will interoperate. It is, therefore, often a major challenge for applications developed in different languages to work together effectively.

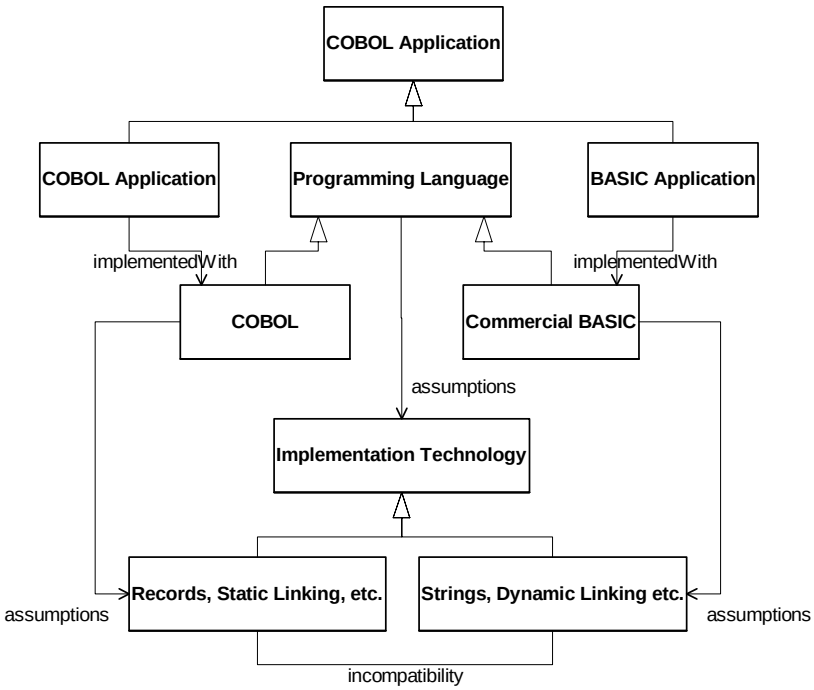

Figure A.13.1 – Tower of Babel

Impact on Legacy Systems: Legacy systems resist new requirements. To circumvent this, new functionality may be either highlighted in an existing but separate application, or developed separately from scratch. If this external functionality is developed using a language different from that in which the legacy system is developed, Tower of Babel may prevent the legacy system from interoperating with it. Additionally, it may become impossible to replace particularly petrified parts of the legacy system with externally developed and superior substitute components written in other languages, making a legacy system an all-or-nothing choice. Finally, if several complementary legacy systems cannot inter-operate, the likelihood that a customer will abandon all of them in favour of a cohesive application suite increases considerably.

Known Exceptions: A completely standalone application clearly has no need to inter-operate with other applications and hence is not affected by Tower of Babel. However, we have seen at EDP that the commercial acceptability of standalone applications is diminishing quickly, and more and more applications are now seen as impaired by the Tower of Babel.

Example: EDP supports a range of applications written in a variety of languages. Many of these were developed in-house, whereas others were absorbed through corporate mergers and acquisitions. Currently, EDP developers are working in Java, Perl, C++, COBOL, Commercial BASIC, Visual BASIC, and a number of other languages. Managing all of these applications and their interoperation is a major headache for EDP. For example, EDP's principle sales order processing system interoperates well with only a rather dated in-house financials package. A recently acquired third-party financials package offers significantly improved functionality but has proven extraordinarily difficult to substitute in place of the in-house package, in large part due to the different choice of implementation language (the former is written in Commercial BASIC, the latter in Visual C++). As another example, EDP has two separate pricing models, one implemented in Commercial BASIC, the other in COBOL. It has been impossible for customers to pick and choose between the two, since interoperability with the particular sales order processing component they have chosen dictates which pricing model they must take. The lack of interoperability and substitutability stemming from Tower of Babel, then, is hampering EDP's ability to offer best-of-class features for some applications, and is hampering satisfaction of a customer's right to choose which components they believe to be right for them. EDP wishes to migrate many of these applications to newer programming languages (particularly Java). Unfortunately, this becomes an essentially all-or-nothing affair since the inability of most of the currently

used languages to interact with Java prevents incremental migration. Consequently, EDP has up to now found legacy system migration to different languages too daunting to contemplate.

A.14. Trial and Error

Name: Trial and Error

Type: Petrifying

Summary: Failure to learn from experience means that we are doomed to repeat the mistakes that have already been made.

Antidote: Past Masters (p.263)

Symptoms: If we keep making the same mistakes over and over again, either individually or as an organisation, we may well be committing Trial and Error.

Causes: Experience teaches us what works and what will not work. When we fail to learn from past experience what has already been shown to work, and what has already been shown to fail, we will continue to make the same mistakes again and again. "If we do not learn from the past, we are doomed to repeat it". Trial and Error is where developers constantly work things out for themselves without reference to their own project history and the project experiences of others. This could in a sense be thought of as a sub-pattern of Reuse Abuse (p.205), but the problem in this case is not poor reuse of artefacts, but poor reuse of experience.

A climate of Trial and Error is often perpetuated by a secretive culture with closely guarded knowledge in the heads of only a few venerated "gurus". Neophytes must bow down to the wisdom of the masters, and are forced to continually stagger in the dark on their own in the hope that they will ultimately prove worthy of revelation of secrets from the inner sanctum.

In extreme cases, there may be a company wide (perhaps tacit) belief that sharing experience to discover best practices is irrelevant, since development projects tend to work out well eventually in the end: “even if it goes wrong at first, it will all come out in the wash”. Comfort with Trial and Error may lead stealthily to a culture of Nil Desperandum [Brown et al. 1999], wherein nobody seems concerned about the increasing incidence of general project dissatisfaction or failure.

Impact on Legacy Systems: Petrification is a gradual drip effect, wherein we keep eroding adaptability little by little until eventually the system is unmaintainable. Those gradual drips are manifest in petrifying patterns, which experience should teach us to avoid. Unless we learn to identify and steer clear of petrifying patterns, and lean

towards productive patterns, we will perpetuate petrification in future development work, and hence will never achieve continually adaptable systems.

Known Exceptions: The possible occasion when appeal to Trial and Error is valuable would seem to be speculative development in some previously unexplored area, wherein a little risky exploration may lead to substantial new breakthroughs.

Example: A major problem at EDP is the lack of sharing of experience within a single project and across multiple projects - developers are thrown in at the deep end where they are expected to "sink or swim". There is almost a sentiment that "I learned it the hard way, so can they". Rather than accepting that a developer in trouble is in need of guidance and training, there is a general tendency to label some people as hopeless. Consequently, the same petrifying patterns are applied time and time again, despite the fact that several experienced developers are well aware of them. Perpetuating this at EDP is the lack of an underlying process to share expertise and the lack of a reward system for doing so.

A.15. Ugly Duckling

Name: Ugly Duckling

Type: Petrifying

Summary: An archaic “green-screen” user-interface makes an application appear dated, is off-putting to users due to its dictatorial look and feel, and is frequently associated with intertwined user-interface and application logic that reduces program comprehension and maintainability.

Antidote: Lipstick on the Pig (p.257)

Symptoms: An Ugly Duckling is a dated, non-pleasing user interface, often requiring non-intuitive interactions and typically tied to a "dumb" terminal. Such interfaces both appear old-fashioned to the user (the "look") and generally dictate a strict hard-coded program-driven rather than user-driven sequence of interaction (the "feel"). Now that most users are experienced with Microsoft Windows or Apple Macintosh applications, they generally demand a more contemporary and intuitive user interface.

Causes: Many applications were developed when dumb terminals were the norm and thus have primitive "green-screen" interfaces.

Impact on Legacy Systems: Primitive user interfaces make a system look old, and will thus reduce its appeal to potential customers. More importantly, though, the issue is not entirely aesthetic: user interface code in green-screen-centric systems is typically intertwined within application logic, complicating the flow of that logic and thus impeding program comprehension and maintainability. That is, the intertwining of code encourages system petrification.

Known Exceptions: The only users we have seen who actually prefer dumb terminal interfaces are those "at the sharp end", where they interact with the system frequently and at high speed. Examples are those who "bang in orders" for telephone customers, and also (some) help desks wherein the pace is hectic. These users frequently prefer a simple green-screen keyboard-driven interface rather than a primarily mouse driven one. Such people are system experts, and to them time is money. When orders or queries are coming in thick and fast, they want to utilise rapid (keystroke-driven) system shortcuts: they generally find verbose narrative and long-winded interaction sequences irritating and cumbersome.

A second (but now rapidly dwindling) exception is where huge numbers of users are already using dumb terminals, and the business cannot absorb the substantial replacement cost involved in switching them all over to PCs or Workstations. Plummeting hardware costs, and the compelling lure of better software is reducing the resistance of all but the most hardened Legacy Customers (see the petrifying pattern of the same name, p.199).

Example: Many of EDP's legacy systems are bound to dumb terminals. This is not surprising, considering some of those systems have grown over decades of development, where dumb terminals were the norm. Many customers already using these systems are entirely satisfied with the traditional user interface, and there has been very little noticeable push from them to move to a more contemporary look and feel. This has resulted, on occasion, in an unwitting complacency at EDP towards dumb terminals. Unfortunately, potential future customers have generally been far less enamoured with the green screen tradition. Indeed, at a recent trade show, those systems with contemporary user interfaces attracted considerable attention, whereas "better" systems with aging interfaces attracted no interest at all. Without pizzazz, even highly functional systems are generally overlooked. The commercial imperative alone, then, demands a user interface overhaul.

A.16. Us and Them

Name: Us and Them

Type: Petrifying

Summary: When technologists and business people do not cooperate with one another, there can be little hope that technical solutions will satisfy business needs.

Antidote: Brothers in Arms (p.226)

Symptoms: Us and Them may exist where technologists and business people rarely talk to one another, where new software features appear to be added on developer whim and have no justification in terms of business needs, and where customer change requests rarely materialise in later software releases.

Causes: An Us and Them culture results essentially from an us-and-them mentality. Business people may see technologists as maverick "geeks" with no commercial sense whatsoever. Technologists, on the other hand, may see business people as "all talk and no substance". Since these groups effectively speak a different language, it is not hard for them to become tribal ("Birds of a feather flock together"). This impairs communication, cultivates mistrust and suspicion, and undermines focus upon the satisfaction of customer needs. The problem is often compounded by the two groups having goals that are at odds with one another. Business people need product immediately to satisfy customer demands, and the "constant failure of developers to deliver on time" is seen as leading to customer resentment and possible loss of future custom. On the other hand, the constant demand from business people for new features "yesterday" puts developers under unrealistic time pressure, making late products even later. Sometimes, unfortunately, management either explicitly or implicitly encourages such a culture, in a variation of Mushroom Management [Brown et al. 1998], by either rewarding it (e.g. counting lines of code, rather than assessing business value delivered) or punishing collaboration (e.g. “get out of the developers hair, they’ve got code to ship”).

Impact on Legacy Systems: Legacy systems resist support for changing requirements. If developers are not kept aware of shifting requirements, and their relative importance, it is highly unlikely that the architectures of information systems will match the real commonality and variability needs inherent in those changing requirements. Consequently, the information systems will fall further and further away from potential support for genuine business needs. This, of course, compounds the problem, so that the

backlog of resisted requirements continues to grow, business people see developers as being less and less productive, and developers see business people as progressively more demanding and more desperate. This exacerbates mutual mistrust and thus perpetuates and strengthens an Us and Them culture.

Known Exceptions: Even though Us and Them is generally damaging, its opposite "All Friends Together" can be equally detrimental, resulting in too much deliberation and not enough action, or with such fickle business demands that developers can never sink there teeth into something substantial. We must, in effect, balance the conflicting lessons from two proverbs: "Many hands make light work" and "Too many cooks spoil the broth". Although we need to be responsive to business needs, there are times when developers need to circle their wagons and get on with the task at hand without constant disruption. This can be particularly true for speculative development projects, wherein often there is not currently an established customer base from which to receive feedback. Here developers need sufficient leeway to explore new ground, without the need for continual business justification, in order to come up with something groundbreaking.

Example: At EDP, there is a very clear distinction between whether you are a technical person or a commercially aware business person. Each group refers to the other group with somewhat derogatory slang: technologists refer to the business people as "big wigs" or "big knobs", whereas business people refer to technologists as "rocket scientists" or "propeller heads" or, rather amusingly, "toupees" (i.e. "little wigs"). There is a definite tension and a considerable lack of communication and understanding between the technologists and the business people. This seems to stem from some belief that "the other side don't get it". The technologists view the business people as technical ignoramuses, bluffing their way through a world they don't comprehend. The business people, on the other hand, seem to view the technologists as prima donnas, exploiting their personal marketability to enjoy a privileged existence with little thought for practicality.

This us-and-them mentality has resulted in a noticeable undercurrent of mistrust between the groups. The unwillingness of each side to respect the perspective of the other has resulted in frequent misunderstandings of the relationship between business needs and technical capability, so that meetings between the two groups have been described on occasion as "the blind leading the blind". This has had the unfortunate

consequence that the two groups frequently "go off in different directions", without a shared commitment to alignment of technology with business needs.

As an example of the difficulties stemming from Us and Them, two of EDP's software developers decided to attempt to break the cycle of mistrust and volunteered to "sit in" on a meeting with a customer where business needs were discussed. Part way through the meeting, a member of senior management (a "big wig") turned to the developers and asked them "what are you doing here? You've got deadlines to meet. Get back to cranking code". Despite this reaction, the developers were actually praised by the customer, and later even by some "big wigs", for their contributions to the meeting, and "finally clearing up stuff we never really understood". The developers noted that they themselves had learned first hand, and for the first time, what the customer really wanted and that this new understanding conflicted considerably with what they had been led to believe previously. As a consequence of this one meeting, the developers re-architected their software to be flexible to the customer's real requirements. Without this meeting, development would have progressed for several months in a direction that conflicted with real customer needs. Despite this demonstration of the clear benefit of their involvement in such meetings, the developers kept away from future meetings with customers and business people for fear of being "shouted at" again.

Appendix B

Productive Pattern Catalogue

The nature of the sickness found, Ulysses,

What is the remedy?

- William Shakespeare, *Troilus and Cressida* I, 3

Productive Pattern Form

Pattern Name

Name: Pattern Name

Type: Productive

Summary: A brief overview of the essence of the pattern.

Target: The petrifying pattern for which this is an antidote.

Features: A detailed description of the pattern and how it manifests itself in a system.

Benefits: The beneficial effects of this pattern on a system.

Impact on Legacy Systems: How this pattern contributes to the reintroduction of adaptability into a petrified legacy system.

Known Exceptions: Instances when the presence of this pattern may not actually be beneficial.

Example: A concrete example of the existence of this pattern in a real-world system, and the positive impact it had on that system.

Alternates: Other approaches to achieving similar benefits.

B.1. A Spoonful of Sugar

Name: A Spoonful of Sugar

Type: Productive

Summary: Keeping Legacy Customers sweet by offering compelling incentives to move forward can encourage them to let go of the legacy systems that hold us back.

Target: Legacy Customers (p.199)

Features: "A spoonful of sugar helps the medicine go down in the most delightful way." We can keep Legacy Customers sweet by offering them some substantial incentive to move forward with us. For example, we can offer them hardware upgrades at a substantially reduced cost, software upgrades and training at no cost at all, and direct contacts with nominated support personnel to deal with any problems they face. In essence, we talk to them, find which sweeteners appeal to them, and then we bend over backwards to encourage them to let go of their affection for the status quo.

Benefits: Keeping Legacy Customers sweet ensures that we don't lose their business and their hard won loyalty, yet also encourages them to move on in the direction that newer customers are taking us. This prevents us from being pulled in different directions by two sets of customers, enabling us to focus upon a single direction wherein all customers are essentially satisfied with the direction we are taking.

Impact on Legacy Systems: Tackling legacy system petrification requires considerable energy and focus, and eliminating the drain of Legacy Customer demands can provide a boost to both. Incentives to move Legacy Customers forward prevent them from holding us back. This gives us greater leeway to tackle the petrification underlying our legacy systems and frees up resources to migrate towards common adaptable systems to which our whole customer base subscribes.

Known Exceptions: There will be some Legacy Customers for whom almost no amount of sugar will be sweet enough. For some customers, the price and their level of resistance may be simply too high for us to bear. In these cases, where the "carrot" hasn't worked, we may need to turn to the "stick". Here, a number of the alternates detailed below may prove effective.

Example: As an example, EDP has a particularly large customer with whom a long-term relationship has been established. To encourage that customer to migrate to newer EDP offerings they have been given a five star service all the way. That customer

receives heavily reduced pricing, immediate on-site service, a direct line to the managing director of EDP, and a considerable early influence over many of the features that appear in future products. These sweeteners have turned this customer into a keen early adopter of EDP's evolving technology, resulting in an excellent large reference site towards which potential new customers can look when considering technology purchases.

Alternates:

There are several alternates to sweetening the Legacy Customer, but most of them involve facing the possibility that we will lose that customer for good. This is obviously only viable if we can compensate for the loss of revenue and loyalty via new business gained or increased satisfaction of the needs of other established customers. Sometimes, we need to "lose a finger to save a hand".

1. Arm Locks

We can add proprietary features to the system that "lock" the Legacy Customer into our products, so that it becomes extremely painful for them to move elsewhere. If we then "drop" support for older versions of our products, the Legacy Customer then has little room for maneuver, and must move forward with us or undergo the greater pain of moving to another supplier with incompatible product offerings.

2. Paying the Piper

Legacy Customers are only a problem when the cost of supporting them is greater than the revenue they generate. One possibility, then, is to negotiate with them an increased annual maintenance fee to cover their maintenance costs. When Legacy Customers are no longer a financial drain, it may even make sense to spin off a separate business to support them. The remaining business can then focus upon moving the product forward to satisfy the evolving needs of the remainder of the customer base and to attract new customers to the product.

3. Pass the Parcel

We may be able to "sell" the support licenses (and consequent recurring revenue) for the Legacy Customer base to a competitor. The term "sell" is used loosely here, since in some cases there may not be much value to extract for them. In some extreme cases, we may even need to pay an external organisation to take over support. The question is one of balancing the lost revenue against the potential benefits gained from being unencumbered by the burden of backward compatibility.

B.2. Babel Fish

Name: Babel Fish

Type: Productive

Summary: Utilising a framework that hides implementation-language differences allows language-transparent interoperability across applications written in diverse languages.

Target: Tower of Babel (p.209)

Features: Customer demands for interoperability and substitutability mandate an "opening up" of applications - enabling them to collaborate both with one another and with the customer's own software packages, irrespective of the languages in which they were developed. The Babel Fish pattern achieves this by advocating the construction of a Babel Fish for each application. A Babel Fish is a language- and location-transparent universal translator, which ensures that all incoming messages are in a form meaningful to the language in which the recipient is developed. Applications communicate via their Babel Fish, which shield the applications from differences between implementation languages.

With a Babel Fish, each application has a symbolic name, which a common Babel Listener maps (via a configuration file) to a network address and an application pathname. All Babel Fish provide services with which applications can then send messages to one another knowing only their symbolic names. Similarly, any application can listen for incoming messages and receive them (via its Babel Fish) in a form meaningful to the language in which the application is written.

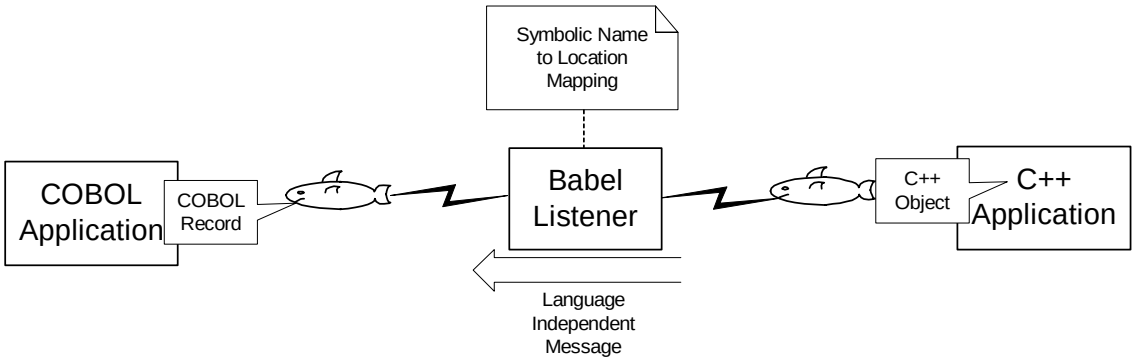

Figure B.2.1 – Babel Fish

At EDP, Babel Fish have been implemented so far to talk between applications written in Java, C, C++, COBOL, BASIC, and PERL. To allow additional programming

languages and applications to be utilised in the future, it is important that Babel Fish are easy to develop. Hence, a Babel Fish Framework has been developed via which new Babel Fish can be constructed with as little effort as possible.

Note that the Command pattern [Gamma et al. 1995] is a helpful complement at the ends of a Babel Fish since it encapsulates each request as a local object, bringing many potential advantages, such as request inheritance, logging, rollback, and so on.

Benefits: With a Babel Fish in place, an application should be able to talk to applications written in other languages, developers have greater freedom with respect to the languages they use, and customers are better able to substitute compatible components.

Impact on Legacy Systems: A Babel Fish enables legacy systems to talk to other applications. This allows us to pick-and-mix the parts of each application that we wish to utilise. Thus, we can eliminate a particularly petrified part of a legacy system and replace it with more effective external functionality in some other system. Furthermore, if the legacy system is developed in a language we no longer wish to continue using, we can now develop new functionality in a language we prefer. In both of these cases, then, rather than reducing petrification, we can bypass it. Finally, Babel Fish facilitate incremental migration of a legacy system to an adaptable replacement, by allowing the dwindling not-yet-migrated legacy code to work as a whole with those parts of the system that have already been migrated and are written in a (possibly) different language.

Known Exceptions: Adapting legacy systems to utilise a Babel Fish takes time. Nevertheless, we have found few circumstances where the approach hasn't paid off. The only circumstance we have found where a Babel Fish cannot work is in languages that have no means via which to talk to a network (although in these cases a single-machine Babel Fish may be possible). One other potential problem is that each application using a Babel Fish must reside in a different address space (e.g. a separate process), since all potential recipients must be listening for messages simultaneously. Communication across address spaces obviously has a performance impact. Since applications tend to collaborate at a high-level interface, we have not found the communications overhead overwhelming at EDP. However, for applications where peak performance is critical, it may be necessary to adopt some alternate shared-memory scheme instead.

Example: At EDP, a number of legacy systems are being incrementally migrated to Java-based replacements. One such system is currently written in Cobol, another in

Commercial BASIC. Neither of these languages can interact easily with Java. Consequently, we have attached a Babel Fish to each application, enabling their interaction. In addition, by using Babel Fish, we have been able to (thus far, partially) migrate several of EDP's legacy systems to adaptable Internet-Based replacements, each with a (diminishing) legacy system back-end remaining in place to fulfil not-yet-migrated functionality. Furthermore, it has been possible to pick-and-mix the remaining back-end pieces across several legacy systems, since the replacement application is blissfully unaware of the language or location of the pieces to which it is talking.

Alternates: At EDP we investigated several alternatives to Babel Fish as a way of achieving language- and location-transparent application interoperability and substitutability. These alternatives were:

1. Intermediate Persistent Storage

One of the main difficulties with divisive languages is that they do not have a common data space. This prevents them from sharing data. One very effective and traditional solution to this is for them to communicate via a database or filing system, which acts as an intermediary between them. Thus, one application writes data to some intermediary storage, then invokes another application which picks up that data, manipulates it, and then repeats the process in the reverse direction. This, of course, can only work where the applications share some common intermediate storage mechanism (which is increasingly likely with the prevalence of SQL [Date and Darwen, 1997], and particularly ODBC [Signore et al. 1995]). There are, though, two downsides. The first is that the applications' internal data must be continually serialised and de-serialised to and from the format required by the database. This is both complex and cumbersome for non-trivial data structures. The second downside, of course, is the significant performance penalty incurred by the (unnecessary) data persistence implicit in databases and filing systems.

2. Ad-Hoc Connections

Traditionally, EDP has addressed Tower of Babel via the development of ad-hoc one-to-one connections between pairs of applications. Maintenance, development, and use of such ad-hoc connections have proven both frustrating and time consuming. The number of ad-hoc pair connections quickly gets out of hand, and as new applications are developed or acquired the number multiplies rapidly. The result is often a cumbersome Stovepipe System [Brown et al. 1998]. Experience has taught EDP that ad-hoc connections are not a viable long-term solution. In the short term, however, or in cases

where a company has only a very small number of applications that require interoperation, ad-hoc connections may be an appropriate quick-and-dirty solution.

3. CORBA

Early on in this project it was thought that CORBA [Henning and Vinoski, 1999] would provide an effective solution to application interoperability problems. Hence, work began on developing CORBA IDL specifications for each of EDP's applications. The realisation was that each application would require only a single IDL specification, leaving transparency issues to CORBA. Unfortunately, this approach introduced two considerable problems of its own. The first problem was that developers of the various applications would need to become expert in CORBA technology - a considerable learning curve which EDP was unwilling to undertake. The second problem was that a number of the languages used by EDP do not have OMG-sanctioned IDL mappings (e.g. BASIC and PERL). This mandated the construction of proprietary adapters between CORBA and these languages. These adapters, it transpired, became rather complicated, and essentially undermined the advantage of using CORBA in the first place. Consequently, CORBA was abandoned and in its place the Babel Fish Framework was developed. Even though CORBA proved unsuitable for EDP, it has the major advantages that it is well supported and well documented. For companies willing to undertake the steep learning curve, and able to confine their development work to only CORBA-supported languages, CORBA may well prove a wise choice.

3. .NET

.NET, from Microsoft [Microsoft, 2001], offers great potential to address the problems described here. .NET advocates a common intermediate language into which diverse high level languages compile. That intermediate language is later compiled into machine code. Code written in any of the supported high-level languages should then be able to interoperate seamlessly. This approach holds great promise. At the time of writing, though, this technology was immature and only available in pre-release form.

B.3. Brothers in Arms

Name: Brothers in Arms

Type: Productive

Summary: Unite developers and business people around negotiating and committing to the highest priority business value.

Target: Us and Them (p.216)

Features: Upper management needs to stimulate, encourage, and coordinate a common pivot point around which developers and business people can work together. In particular, compatible reward systems need to be put in place to ensure that both groups have incentive to cooperate rather than wrestle. Unless there is a shared commitment to a common understanding of realistic goals centred around negotiation of prioritised customer needs, it is highly likely that the computer systems being developed will continue to fail to support those needs adequately. Hence, business people need to be rewarded for identifying and prioritising the key business value that needs supporting and, perhaps more importantly, for getting out of the way of developers and freeing them from lower priority work. Similarly, developers need to be rewarded for delivering product that continually delivers the highest business value into customer hands. This shared focus upon identifying and delivering highest business value gives business people and developers common ground on which to come together.

Benefits: One of the most de-motivating situations for developers is uncertainty about what they should be working on - either because their goals are so poorly defined, or because they cannot see value in the work they are doing. De-motivation is devastating for productivity. Negotiating with developers clearly defined goals with stated high business value gives them both something to sink their teeth into and a sense of urgency to keep them going. From the perspective of business people, they have the great advantage of now determining and prioritising precisely what developers are working on. If developers end up working on the wrong thing, business people have only themselves to blame. From the customer's perspective, the product they receive will always be that which delivers the highest business value currently possible. In this context, developers essentially see business people as their proxy to the customer. Their relationship with business people, therefore, may also benefit from appeal to proven customer interaction patterns [Rising, 2000].

Impact on Legacy Systems: Getting developers and business people focused on delivering prioritised business value gives very clear directions in which to take a legacy system. One problem with petrified systems is that they often proliferate in a culture of almost random flapping about, wherein a flurry of activity starts adding a feature but momentum dies out when the difficulty of adding that feature becomes evident. In a panic-stricken effort to deliver at least something, a seemingly random new feature is grasped at and another flurry of doomed activity begins. Instead, always focusing both developers and business people on negotiating and committing to high business value ensures that motivation comes from all angles. This keeps momentum going when addressing even the thorniest petrification for the introduction of important (high priority) business features.

Know Exceptions: Some speculative development work can actually be impaired by close attention to immediate business need. Radical breakthroughs requiring inspired insight can be stifled by too much "interference" on the creative freedom of developers. Sometimes, developers are best left alone to come up with something completely unexpected. This is particularly the case in a research environment. However, taking the results of that research into something practical probably requires a shift again towards developers and business people working hand in hand.

Example: We have found it extremely hard to break the Us and Them culture at EDP, where we have taken an active role in encouraging communication and collaboration, working as far as possible as a mediator between the groups. Our progress in breaking down mistrust and building rapport has been noticeable but very slow. We have had some success in specific cases, wherein highly motivated developers have sought out business people and almost begged them for collaboration on feature prioritisation. Hence, we have seen first hand the benefits of collaboration and negotiation between some commercial and technical groups, but have failed to come up with a company wide cultural shift. The underlying problem at EDP is both a barrier of mistrust on each side and a tacit management belief that talking detracts from progress. Unfortunately, without strong and ongoing support and reward from upper management, changing the corporate culture has proven to be major challenge.

Alternates:

1. Butt Kicking

Business people may feel developers have "let them down" so often in the past that there is no point talking to the developers directly, and instead it is often more effective

to coerce a higher level of management into getting things moving. Here, the "big boss" can "kick those developers in the butt" and have some newly promised feature given immediate high priority. When this proves effective (even at the expense of other features) it is seen as evidence that developers were indeed "holding back" after all, and "butt kicking" will be appealed to as an effective productivity booster again in the future. Developers, however, resent such mistreatment, since they well understood that other important features have now been forced into artificially lower priority, which will only lead to subsequent complaints when those features are delivered late. Indeed, developers may be tempted to ask upper management to "butt kick" the business people right back for requesting "dumb" features that can only delay some high priority project which has already been sanctioned "from on high". Butt kicking can certainly work in the short term. Unfortunately, we have found this game of vicarious "butt kicking" Ping-Pong tends, in the long run, to be counter productive since it perpetuates the resentment, mistrust, and tribalism underpinning Us and Them.

2. Rounded Individuals

An appealing alternative to Brothers in Arms is elimination of categorisation by “rounding out” developers with business skills, and business people with technical skills. The idea being that we will be blessed with technically savvy, customer-friendly, “renaissance men”. Despite this appeal, we have rarely found people able to master both sets of skills well. Rather, the result is often a mediocre developer with shallow commercial awareness. Nevertheless, where such a rare individual can be found or cultivated they would be an excellent peace-broker with a foot firmly placed in each camp.

B.4. Dynamic Pluggable Factory

Name: Dynamic Pluggable Factory

Type: Productive

Summary: By developing a framework for object-creation factories bound by a dynamic linker we enable new object types to be linked into an application at runtime.

Target: Bound and Gagged (p.186)

Features: Dynamic Pluggable Factory [Lauder, 1999c] has quite a long ancestry. It is a highly flexible variant of the Named Product Pluggable Factory pattern [Lauder, 1999c; Vlissides, 1999], itself a variant of the Pluggable Factory pattern [Vlissides, 1998; Vlissides, 1999], which is in turn a variant of the Abstract Factory pattern [Gamma et al. 1995]. The intent of any factory is to produce products. In the standard Abstract Factory pattern, the types of concrete product (i.e. object) that can be created are hard coded to specific concrete factories. The Pluggable Factory pattern (see Figure B.4.1) enhances this to runtime variation of the product type that a specific factory can create, but the total set of product types is still hard coded. With the Pluggable Factory there is only one factory, which is assigned at runtime with a prototype for one concrete subclass of each of the abstract types of product that can be created. Varying these prototypes allows the factory to create products of different concrete types.

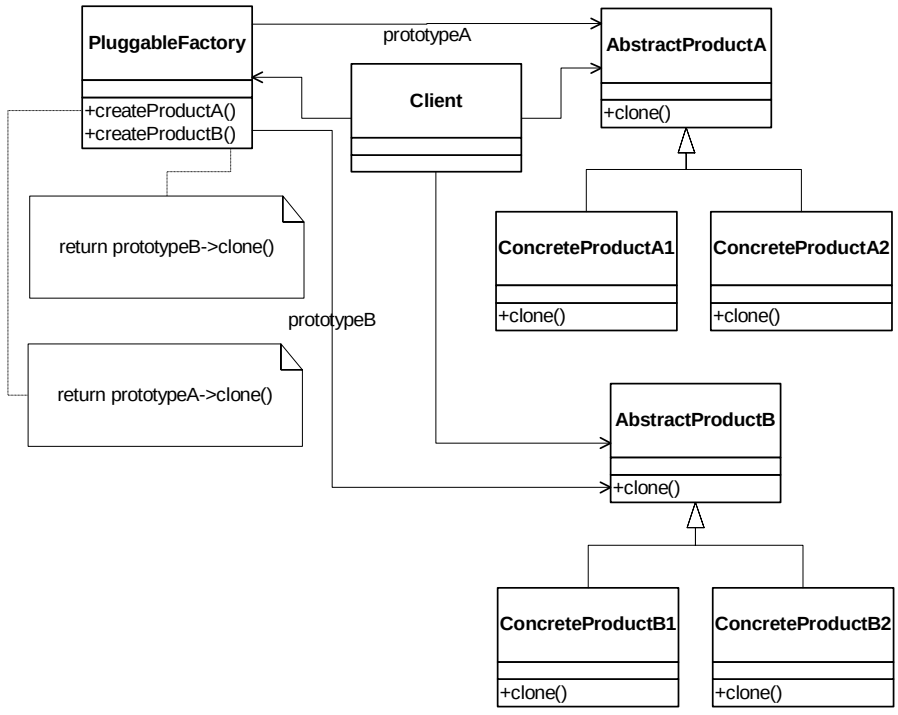

Figure B.4.1 – Pluggable Factory Pattern

The Pluggable Factory, then, can only create one concrete type of product for each abstract product at any given time. The Named Product Pluggable Factory generalises this further to allow runtime naming of the product types required from each factory. Here, as depicted in Figure B.4.2, multiple prototypes are retained (one for each concrete product), each associated with a symbolic name representative of the relevant concrete class. This enables us to create all of the available concrete product types by specifying a symbolic name for the prototype to be cloned, but again the set of available product types is still hard coded.

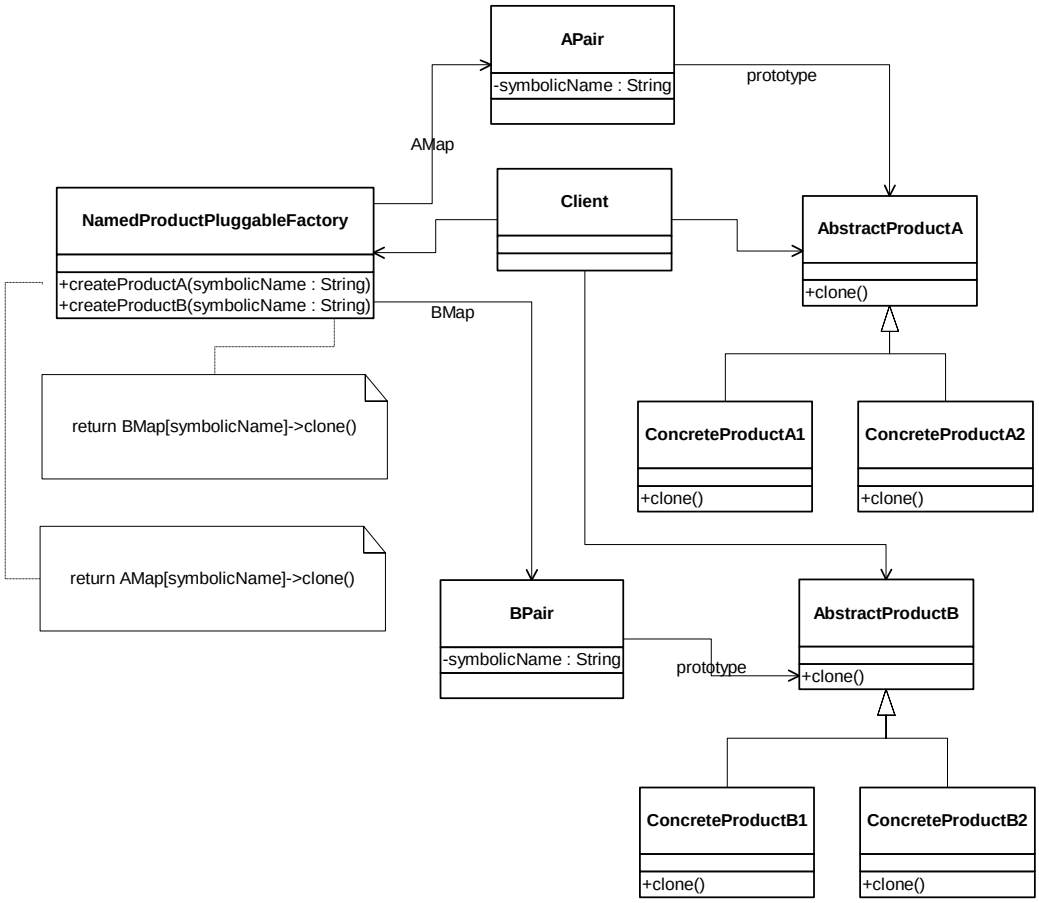

Figure B.4.2 – Named Product Pluggable Factory

Dynamic Pluggable Factory extends Named Product Pluggable Factory to allow runtime naming of products for factories where the names and types of those products did not necessarily even exist when the application was written.

Unlike its predecessors, the Dynamic Pluggable Factory contains no concrete product types whatsoever. Instead, the Dynamic Pluggable Factory is responsible for determining concrete product types itself. To achieve this, applications request products of a given type by passing a symbolic name (a string) to a creation method in the Dynamic Pluggable Factory. The range of permitted symbolic type names is held in an

external configuration file shared by the application and the Dynamic Pluggable Factory. The Dynamic Pluggable Factory looks to see if it already has a prototypical instance of a concrete product type associated with the given symbolic name, and if so clones that prototype and returns it. Otherwise, the factory "picks up" a prototypical instance of the appropriate product type from a location specified in the shared configuration file, clones that prototype, returns the clone to the application, then retains the prototype for subsequent cloning when the same symbolic type name is requested.

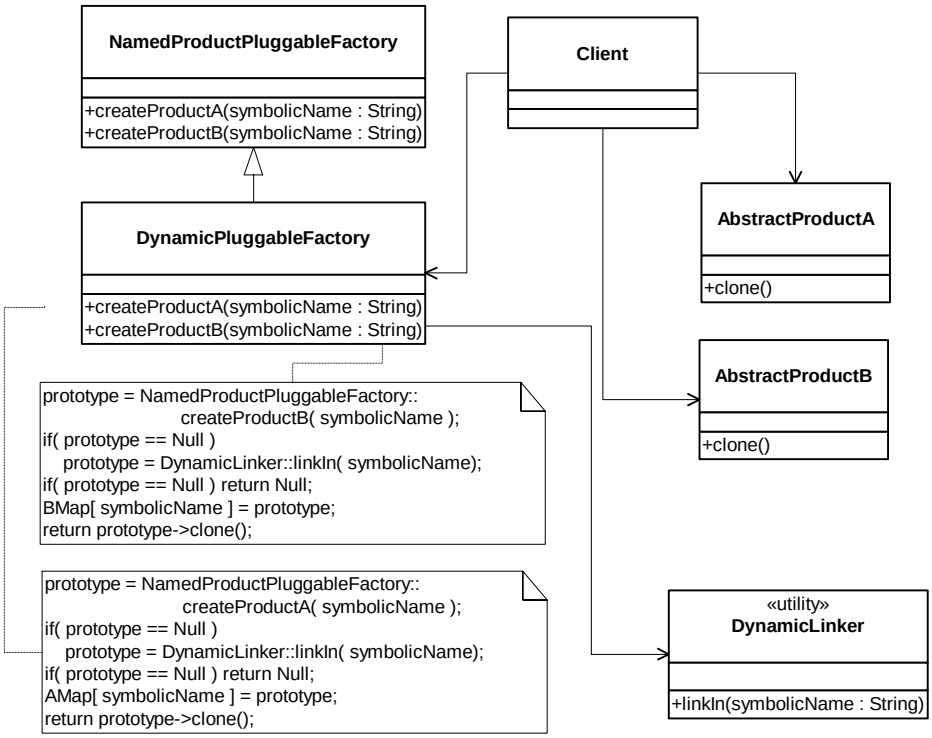

Figure B.4.3 – Dynamic Pluggable Factory

How does a Dynamic Pluggable Factory “pick up” a prototype from the location specified in the configuration file? The secret lies in utilising the operating system’s dynamic linker. A dynamic linker allows a running application to link new symbols into itself at runtime. Particularly useful is that an application can use the dynamic linker to locate and link-in a new function by specifying its name as a text string. The application is then able to use the newly dynamically linked symbol as if it had been part of the application all along. Now, most dynamic linkers cannot handle classes (object types) directly since they are not particularly aware of object-orientation, and they take a very non-object-oriented view of symbols. To load a class into an application directly using a dynamic linker would be extremely difficult. However, this is not our requirement. What we require is a prototypical instance of a class, not the class itself. The solution, as depicted in Figure B.4.3, is to use the dynamic linker to find, for each symbolic product

type name, a simple non-object-oriented function whose sole purpose is to create a prototypical instance of the associated concrete product class. That function can itself invoke an appropriate object-oriented program to perform the actual creation, and the newly created prototype is then passed back to the factory via the non-object-oriented function. Once the factory has received the newly created prototype, it stores and then clones and returns it in response to application requests.

Benefits: A Dynamic Pluggable Factory enables an application to keep running even in the presence of new capabilities being added, and also facilitates local adaptation of the application to the needs of a specific context without access to the application's source code.

Impact on Legacy Systems: Removing knowledge of object types from a legacy system increases the maintainability of that system. In particular, since new object types are added externally to the core code, there is no possibility that adding those types can obscure and petrify the architecture of the core code. Furthermore, since other people are now able to add new object types independently, they do not have to wait for the legacy system development team to schedule their requests into the growing backlog of pending requirements changes. Locally required object types, then, can be added more quickly, implying that the system is evolvable at a rate closer to the rate of requirements change.

Known Exceptions: In applications where required object types do not vary rapidly, this pattern is probably overkill. We saw an instance of this at EDP, where a developer introduced Dynamic Pluggable Factory into a part of the system purely on technical whim. It transpired that object types in that area never varied, and hence the introduced flexibility was superfluous. As another exception, one drawback of Dynamic Pluggable Factory is that the all object types associated with a given factory must share the same interface. In applications where this is not possible, object types must necessarily be hard coded. A final problem we have noticed is that debugging any code that uses dynamic linking is considerably more difficult than with statically linked code, as is telephone support for technical users experiencing problems with their own plugged-in factories. If this support burden is unacceptable, the flexibility introduced by the pattern may have to be sacrificed and simpler static code adopted instead.

Example: This pattern is described further in [Lauder, 1999c], which presents a detailed example of the pattern in use in a Banking system (where application downtime is unacceptable). In addition, we have used Dynamic Pluggable Factory at EDP as an

interface to various filing systems (e.g. ISAM, Unix, Pick, etc), with a consistent interface across them. When a customer needs access to a further filing system type, it is added as a dynamically loadable object type, without any need to change EDP's own application code. As a final example, EDP's WebWidgets Framework uses a Dynamic Pluggable Factory to create instances of named website components.

Alternates:

1. Java

Java and a number of other languages simplify the code for Dynamic Pluggable Factory by providing facilities for dynamic class loading within the language itself. This, however, only replaces the dynamic linker code in the pattern, since factory objects are still required to eliminate application dependency on the classes that are dynamically loaded.

2. Interpreted Scripting Languages

If the variations between the required dynamically loaded classes are minimal, it may be appropriate to create a small, interpreted scripting language with which to express those differences. Users could then effectively write their own "classes" in this scripting language, which is then interpreted and executed by the application. This is particularly well-suited to specialised domains, where business experts wish to write localised business rules using terms from that domain's ontology [Uschold and Gruninger, 1996] without resorting to full-blown programming languages. The scripting language, though, must be relatively simple for it to have any advantage over established programming languages in conjunction with Dynamic Pluggable Factory.

3. Open Source

If the problems are mainly to do with the need for user-specified object types, rather than with minimising application down time, it may be possible to release the source code of the application for local adaptation. Indeed, there is an interesting (although to some commercially worrying) trend towards "open" (i.e. freely available, and usually free-of-charge) source code, wherein there is a potential for product quality and market penetration to improve considerably as a result [DiBona et al. 1999]. There is, of course, no reason why an open source policy could not be combined with a "for use" fee. Also, putting support responsibility into the public domain (to be taken up by volunteer enthusiasts, as happens with Linux), and perhaps charging fees for a premium

support service could mitigate commercial worries about the difficulties of supporting Open Source products.

B.5. Explicit Protocol Reflection

Name: Explicit Protocol Reflection

Type: Productive

Summary: Rather than wrap guard code around functions, we can reflect time-ordered collaboration protocols explicitly as a concern separate from functional code.

Target: Code Pollution (p.188)

Features: In response to the Code Pollution problem, we can appeal to a direct realisation of Statecharts [Harel and Politi, 1998] in a component's implementation. Statecharts are a flavour of Finite State Machine [Yacoub and Ammar, 2000]; they express various abstract states [Dyson and Anderson, 1998] in which a component (or fragment of a component) may reside over time, and connect those abstract states via transitions from one to another in response to received events of interest. Thus, the current abstract state determines (via its transitions) which events are currently of interest, what action to perform in response to each event, and which abstract state to mutate to next.

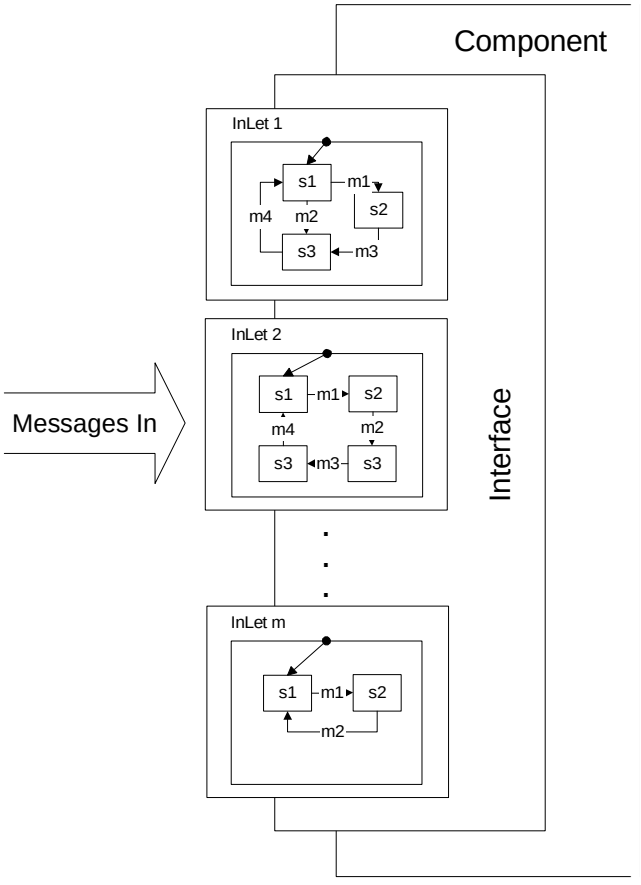

Figure B.5.1 – Explicit Protocol Reflection

In our *EventFlow* model (originally termed EventPorts, and further described in [Lauder, 1999a; Lauder and Kent, 1999a]) we utilise direct realisation of statecharts via their association with *InLets*. The Interface of an EventFlow-based Component exports one or more InLets via which events are received. When an InLet is created, it is associated with a dynamically configurable *Statechart* (see Figure B.5.1, above).

The *CurrentState* of a Statechart determines (via its *Transitions*) which messages the associated InLet is currently interested in receiving. If an InLet receives a message that is not associated with one of the Transitions of its Statechart's *CurrentState*, then that message is simply ignored. If, however, that message matches the interests of a Transition, and any associated *Condition* is satisfied, then the Transition is taken, any associated *Action* is invoked, and the target abstract state of that Transition becomes the new CurrentState.

Benefits: This direct reflection of Statecharts in a component's interface has the potential to eliminate Code Pollution, since the Statechart mechanism itself ensures that an event-induced method cannot be invoked unless the time-ordered protocol inherent in the Statechart is respected. The elimination of Code Pollution simplifies the individual operations of the component and thus eases component maintenance. Furthermore, the fact that the protocol is explicitly recorded in a single place helps humans to comprehend and describe that protocol since it is no longer scattered across diverse guard code. Since a component may have many InLets, and each InLet is associated with a Statechart, each component may exhibit multiple orthogonal Statecharts, the sum of which represents its total abstract state. The set of current states across these Statecharts determines, at any given moment, the total message interests of that component. Each InLet may, therefore, be considered a porthole via which one *facet* of a component's total abstract state is exported. The separation of a component's total abstract state and state transitions into a set of orthogonal facets and associated Statecharts leads to a clean separation of concerns and thus enhances component maintainability.

Impact on Legacy Systems: Eliminating the unnecessary intertwining of the fundamentally independent concerns of business functionality and time-ordered protocol enforcement simplifies the code associated with each concern. Code Pollution contributes to petrification by making code more difficult to understand and modify. Eliminating Code Pollution via Explicit Protocol Reflection, then, contributes to an

increased potential for understandability and maintainability, thus reducing the effect of petrification and easing the ability to reflect ongoing requirements change.

Known Exceptions: We have observed at EDP that some developers find it difficult to work with Explicit Protocol Reflection since they are more familiar with "all the code being in one place". Explicit Protocol Reflection requires developers to maintain two separate concerns (business code and protocol reflection code), and it transpires that this can be an uncomfortable and alien experience for some. This might, or course, be simply a matter of lack of exposure, and perhaps time will ease such discomfort. Nevertheless, there is little point in introducing technologies that the developers at hand find ill-matched with their own personal leanings. A separate issue is that, in many applications, collaboration protocols are very loose, being only minimally constrained. In such circumstances, the benefits seen from Explicit Protocol Reflection will be minimal, since there is little guard code in place to obscure the business code. Of course, it will then be necessary to monitor our caution so that as and when the underlying collaboration protocol is embellished by ongoing requirements change we can recognise the stage at which Code Pollution has become a real problem and Explicit Protocol Reflection should be introduced.

Example: This is the only pattern in the catalogue for which we do not have a real-world example. Consequently, the pattern is tentative. Nevertheless, having “played” with this pattern during research, developers at EDP have been convinced of its validity and, thus, have committed to its deployment during forthcoming migration work.

Alternates:

1. Documentation

One of the main problems with Code Pollution is that collaboration protocols are implicit in scattered code rather than explicit in a separate concern. One simple solution to this problem is to maintain appropriate documentation (in terms of recorded Statecharts) to express the combined effect of the scattered protocol guard code. This will both enable the user of a given component to understand and respect its collaboration protocol, and reduce the intellectual burden on the maintainer of a component (who would otherwise have to disentangle the intent of polluting guard code for themselves). We have certainly found that even where a development group is documentation-light, the maintenance of collaboration-protocol Statecharts pays great dividends.

2. Aspect-Orientation and Subject-Orientation

We have noticed interesting parallels between our own work and that of Aspect-Oriented [Kiczales et al. 1997; Lopes et al. 1997; Lopes et al. 1998; Lopes et al. 1999] and Subject-Oriented [Harrison and Ossher, 1993; Clarke et al. 1999] programming and design. In particular, much AO and SO research is focused upon addressing two major impediments to software comprehension and maintainability: *Tangling* is an intertwining of code or design addressing multiple separate areas of concern, and *scattering* is a dispersion of code or design for a single concern throughout diverse parts of the system. The Code Pollution petrifying pattern can be viewed as a manifestation of guard code *scattered* throughout methods so that it is hard to understand the overall flow of control between components, and *tangled* within the methods so that the guard code itself and functional code is obscured. The corresponding antidote pattern, Explicit Protocol Reflection (as described here), removes scattered and tangled guard code from functional code, and collates protocols in separate concerns (as statecharts). Likewise, the Ugly Duckling pattern (p.214) can be viewed as a tangling and scattering of user-interface code throughout an application. Its antidote, Lipstick on the Pig (p.257), effectively untangles and collates user interface code into a separate concern. As a final example, the Persistent Problems petrifying pattern (p.203) sees tangling and scattering of persistence code as inherently detrimental to adaptability, and its antidote (It's Not My Concern, p.251) effectively pulls that code out, again into a separate concern of its own.

We find these similarities quite encouraging, and we are confident that ongoing AO and SO research will provide us with many fresh insights into the nature of petrification and its resolution. Note, however, that, unlike the AO and SO communities, we are not advocating new programming languages. Rather, we have chosen to work within the constraints of existing languages since pressing commercial needs and real legacy systems drive our work.

B.6. Firewalling

Name: Firewalling

Type: Productive

Summary: Placing a firewall between a newly divided application front end and back end both protects the back end from malicious attack and allows the front end to be adapted to protect the user from back end complexities.

Target: Human Shield (p.195)

Features: We want to allow customers to access the services of the application without passing through a Human Shield. However, opening up an application to access from outside (particularly over the Internet) leads to both security concerns and ease-of-use concerns. Both of these concerns can be resolved by first splitting the user interface from the server-side application (see Lipstick on the Pig, p.257). We then address security concerns by not allowing the front-end to talk to server-side applications directly. Instead, we construct a firewall, wherein all interactions with the system are via messages that have to pass through a security screen. This firewall protects the server from malicious interaction by blocking unauthenticated or out of sequence messages. Once this protection is in place, we can support customer-oriented business processes directly on the server side, and embrace suitable user-friendly scenarios in the user interface.

Benefits: As noted in the features section above, a firewall does not let would be attackers in through the front door, thus protecting the server-side application. An added benefit is that since we have a clear separation of front-end from back-end, we should be able to "plug-and-play" at both ends. Thus, a different server-side application could be substituted, and as long as it adheres to the defined message structure will itself also be protected by the firewall. At the front-end, the owner of the website is free to substitute any front-end they choose, as suits their own business processes.

Impact on Legacy Systems: Opening up a legacy system can be like opening up a can of worms. Legacy systems were often not designed with external users in mind, and hence have neither support for nor protection against them. Applying a firewall provides this protection externally, thus enabling the legacy system to be accessed without fear that its weaknesses will be exposed. This then gives us breathing space, during which we can address the petrification inherent in the legacy system at a more measured pace.

Known Exceptions: Experienced and trusted users may find the security and user interface protection we put in place to be cumbersome and stifling. In such cases, it may be appropriate to provide a trusted user backdoor which bypasses some of these measures allowing unhindered access to the full server application. Furthermore, we have found that even where a firewall is employed in full, people usually still want to know they can get through to a warm blooded human when they need to. Thus, the computer system can become a primary interface, but not the only one. As a minimum, a contact address and telephone number should be in place as a backup for those who require the human touch.

Example: At EDP we have developed and are utilising a messaging facility termed NetEnabler, which was inspired by EventPorts [Lauder and Kent, 1999a], to realise secure system interaction. NetEnabler is a secure message-oriented middleware firewall that fits between collaborating components and ensures that only approved, authenticated, and in-sequence messages pass through. In terms of shielding users from application complexity, we have developed an extensible user-interface product called WebEnabler, which allows user interfaces to be re-created by non-programmers. The website owner is thus free to alter the user interface to support their own business process scenarios.

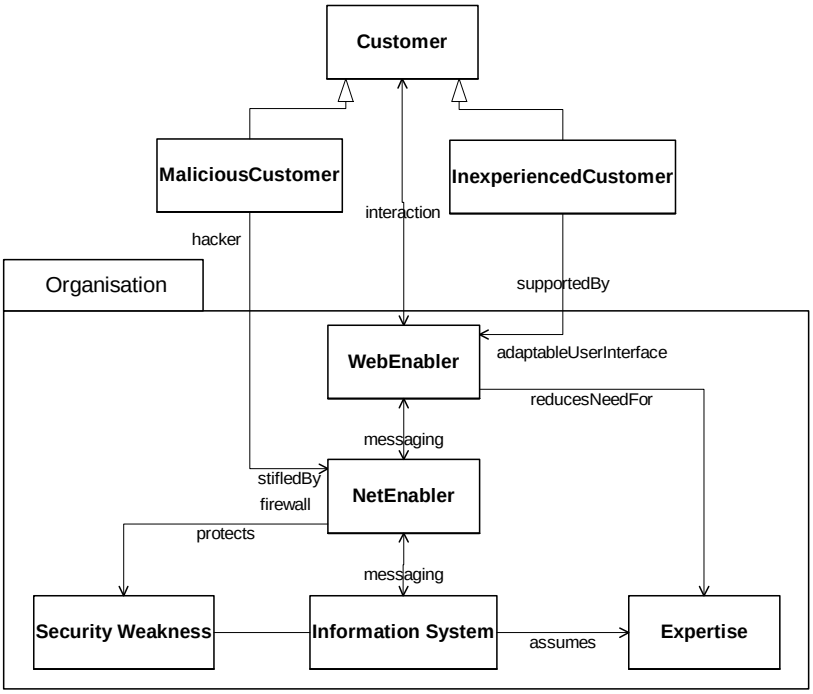

Figure B.6.1 – Firewalling

WebEnabler is underpinned by page fragments termed WebWidgets, the flavour of many of which was inspired by [Nielson, 1999]. WebWidgets encapsulate all supported

dynamic interaction with the server side (via message passing through NetEnabler). To support additional dynamic functionality new WebWidgets can be added (albeit by programmers rather than users) via EDP's WebWidget Framework.

B.7. Gold Mining

Name: Gold Mining

Type: Productive

Summary: When current requirements are lost we must appeal to multiple sources of domain knowledge, following diverse clues, if we are to unravel a competent understanding of the domain being supported by an information system.

Target: Gold In Them Thar Hills (p.192)

Features: Gold Mining is concerned with reconstructing a model of the "lost" requirements for an information system and taking these mined requirements as the starting point for the negotiation of new requirements. The term "model" is used to imply that the eventual aim is to rebuild sufficient business process understanding in the heads ("mental models") of the project team. That is, we are aiming to recapture lost expertise, and expertise rightly belongs in human heads not on paper. However, when rebuilding that expertise, we may choose to construct paper-based domain-explorative models ("sketches" to unravel static views, and "cartoons" to explore system dynamics). See for example [Lauder and Kent, 2000d], which introduces a new form of business-oriented statechart. Furthermore, we may appeal to speculative-archival models ("blueprints" to record static information, and "filmstrips" as in [D'Souza and Wills, 1998], for system dynamics) as a supportive means for transferring expertise (in addition to human conversation) to new team members.

Diverse "clues" in the varied artefacts being mined imply a need for continual negotiation of a mutually satisfactory explanation both for what is being uncovered and how to "fill the gaps". Thus Gold Mining should be seen as a collaborative effort between developers (who can more readily comprehend the code) and business people (who can more readily comprehend the domain). See [Lauder and Lind, 1999; Lauder and Kent, 2000b] for examples of how such collaboration may proceed. The aim is to reach a mutual understanding and commitment to system requirements before undertaking further development work. This mutuality is crucial; without explicit commitment to a shared ontology, different stakeholders may approach requirements negotiation blissfully unaware of one another's (perhaps misguided) perception of the business processes the system is intended to support. Thus, the collaborative process of negotiation underpinning Gold Mining provides a common ground upon which future requirements negotiation can build.

Benefits: Gold Mining should eliminate nasty surprises, in terms of accommodating requested modifications in response to reflecting ongoing business process change. Gold Mining enables us to understand more clearly which business processes are supported and how they are supported, as well as understanding what the impact of future business process change will be on the system. Two things result from Gold Mining. Firstly, we (hopefully) uncover a rationale for the current implementation so that we can better understand which aspects of the system are legitimate (grounded in genuine requirements) and which are accidental artefacts of the history of the system's construction. This knowledge can then feed into our cultivation of a permanently legitimate architecture (see the Legitimate Architecture productive pattern, p.254). Secondly, Gold Mining results in a business-domain oriented ontology (a shared domain understanding, in the spirit of [Uschold and Gruninger, 1996]) that is *consistent* with the behaviour of the information system. It would be wrong to claim that Gold Mining recovers original system requirements, since many original requirements may be lost forever. More realistically, we are often scavenging to uncover a shared understanding of requirements with which all team members are satisfied[60].

A major beneficial "side effect" of Gold Mining is that it can lead an organisation to think about business process change *before* it creeps-up on the business. When an organisation manages it business process change purposefully it will be better able to determine and accommodate the impact of those changes on the information systems that support the business.

Impact on Legacy Systems: It is often the hard-earned reflection of lost or tacit business knowledge that makes legacy systems such a critical issue for an organisation. Indeed, it seems that the real challenge with the development of replacement systems for legacy systems is not the actual coding – EDP developers assure us that they can bang out code in next to no time - but the re-construction, and transfer to the heads of the systems people, of the historically negotiated requirements which the legacy system satisfies. The justified fear is that any replacement for a legacy system may well accurately support the knowledge of which a business is currently explicitly aware, but will fail to reflect – and hence be unable to support - the lost and tacit knowledge implicit in and essential to the organisation's business processes, until that replacement

[60] Indeed, an interesting paper by Randall and colleagues [Randall et al. 2001] observes that "remembering" lost "organisational memory" is rarely the collation of historic facts, but rather a complex and ongoing social process of striving for a negotiated and shared after-the-fact rationalisation of prior and current work.

system has undergone a prolonged and painful debugging process. This will only come about after the replacement has already been deployed and its weaknesses are highlighted (possibly over many years) via its ongoing failure to support users adequately. The organisation has already undergone this pain in the long-term debugging of the legacy system and they do not want to go through it again. Gold Mining essentially uncovers, as far as possible, design rationale for a legacy system. In large part, this helps us to work out which aspects of the system's architecture are an accidental artefact of the construction history (such as dead code, and "hacks") and which are a legitimate reflection of the commonality and variability inherent in the requirements being supported. From this recaptured understanding we can construct an initial legitimate architecture for a replacement system, with a demonstrable rationale for the commonality and variability assumptions it reflects.

Known Exceptions: Gold Mining is really hard work. In fact, we found at EDP that there is often so much rubble among the gold nuggets that it was tempting to call this pattern Gold Panning rather than Gold Mining. Lost requirements are rarely sitting there on the rock face waiting to be plucked by eager hands, rather the "important stuff" is often buried deep within its sources, and we consider ourselves lucky if we ever stumble across it at all. Worst still, we have found there is a lot of "fools gold", where we think we can justify some design rationale but eventually it turns out the design was probably just based on developer whim, or worse still a "neat hack". Gold Mining, then, can take a long time, and often returns little reward for the effort involved. We need to balance the cost against the benefit gained. Ultimately, there is little point in mining beyond the point at which we get something valuable. Importantly, we have realised at EDP that not all lost requirements are necessarily worth recapturing. We should be driven, then, only by what we need to know rather that being seduced by what might still be in there; since we have learned that excessive zeal for Gold Mining only leads to a "mining paralysis" equivalent to the "analysis paralysis" that can befall a new development effort. In summary, Gold Mining is only worth it if the results will pay for themselves, and unfortunately, at this point, we don't know how to make that judgment other than rely on the gut feeling of the team doing the mining work.

Example: Armed with enthusiasm for Gold Mining at EDP, we originally subscribed to a substantial one-shot up-front Gold Mining effort, to capture a full picture of system requirements. In practice we have realised this is woefully unrealistic. Firstly, for all but the smallest systems, the amount of time required to completely "mine the gold" up front is simply commercially unsustainable. Secondly, without an immediate need for a

particular nugget of gold, the process tends to feel rather abstract, with – at least in the EDP experience - a tendency to be performed half-heartedly. We have found that better results (in terms of time and quality) tend to result from opportunistic Gold Mining. More specifically, we have learned at EDP that we should be driven by resisted (usually backlogged) requirements.

When asked to enhance an information system with a new or backlogged requirement, we now first determine the level of the system's resistance to that requirement. Historically, the temptation for a significantly resisted requirement has been to force fit a "neat hack" into the system. We now look at a temptation to hack as an incentive to Gold Mine the requirements whose "loss" leads to our uncertainty about current design rationale. It is this uncertainty that underpins hacking (so we don't "rock the boat" with unanticipated side effects). Thus, we now undertake just enough Gold Mining to rationalise the current system behaviour in the areas overlapping the resisted requirement and then refactor (see Legitimate Architecture, p.254) the system just enough to better reflect the commonality and variability inherent in those overlapping current requirements and the previously resisted requirement we are about to add. Over time, there emerges an overall understanding of the requirements for the most active parts of the system, whereas for those that rarely change we pragmatically "let sleeping dogs lie". A substantial downside of this "just in time" approach to Gold Mining is that, again, we are promoting a "patchwork" understanding of the business processes we are supporting, and this understanding is often very rough around the edges. Our only solution to this at EDP, and unfortunately it is only partially successful, has been to recognise where edges are looking particularly frayed and, when time allows, flesh out a high-level general consensus on how to smooth those edges out. Commercial pressures, though, tend to make such "edge smoothing" all too infrequent.

B.8. Implicit Invocation

Name: Implicit Invocation

Type: Productive

Summary: To eliminate Tight Coupling, abandon explicit invocation, in favour of Implicit Invocation wherein separate parts of the system are decoupled from each other and hence are unaffected by "side effects" resulting from changes to one another's interface, implementation, or even existence.

Target: Tight Coupling (p.207)

Features: We want to eliminate Tight Coupling, wherein conceptually distinct parts of the system are explicitly aware of one another and invoke known operations on one another. To do this we need to abandon the practices of explicit invocation, and instead adopt a model of Implicit Invocation [Shaw and Garlan, 1996] wherein separate parts of the system become decoupled from each other. In Implicit Invocation, instead of invoking operations directly, each part of the system registers its interest in the occurrence of certain events with a shared broker. Next, each part of the system broadcasts a series of events through that broker. The broker then propagates these incoming events onto the parts of the system that have registered interest in them. This is sometimes termed the "push model" of collaboration, wherein events are being pushed out to whoever happens to be listening for them. Explicit invocation, on the other hand, relies on the "pull model", wherein events are "pulled" from a supplier.

Implicit Invocation underlies a number of recent design patterns (Observer [Gamma et al. 1995], Multicast [Vlissides, 1997a], Event Notification [Riehle, 1996], Reactor [Schmidt, 1995], etc.) and a number of commercially available products (Rendezvous [Tibco, 1999], SmartSockets [Talarian, 1999], iBus [SoftWired AG, 1999], etc.). We have implemented an implicit-invocation-oriented technology, originally named EventPorts [Lauder, 1999a; Lauder and Kent, 1999a], now renamed *EventFlow*. A component developed using EventFlow exports an *Interface* and an *Outerface*. A component receives events (message) from other components through its interface and sends events to other components through its outerface. Interfaces and outerfaces consist of dynamically configurable *EventChannels*. As shown in Figure B.8.1, there are two types of EventChannel: interfaces consist of *InLets*, via which a component receives events from other components, and outerfaces consist of *OutLets*, via which a component sends events to other components.

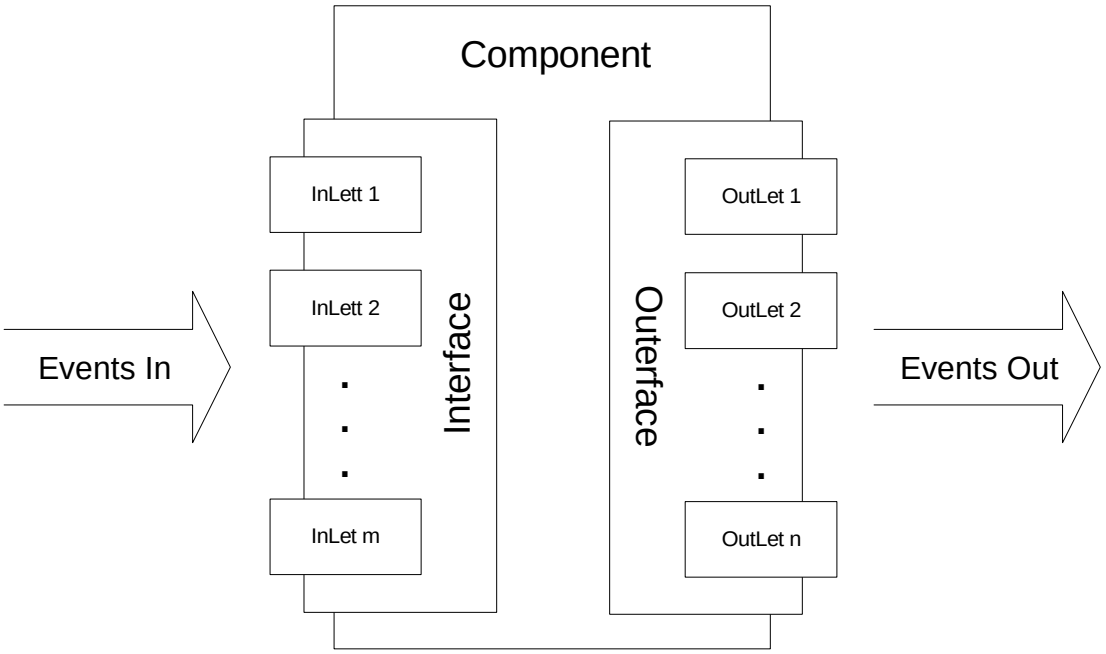

Figure B.8.1 – EventFlow EventChannels

Each of a component's InLets can be dynamically attached to and detached from any number of OutLets on any number of other components at a given time. An InLet registers its event interests with the OutLets to which it is attached. Each OutLet acts, in effect, as a broker that tracks the event interests of attached InLets and receives and forwards messages according to those registered interests (see Figure B.8.2).

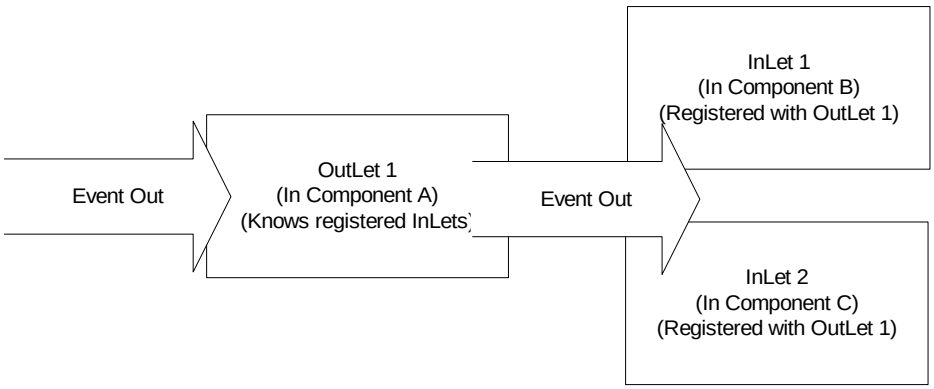

Figure B.8.2 – InLets and OutLets

Benefits: In Implicit Invocation, brokers form intermediaries between collaborating components, resulting in component de-coupling. This reduction in inter-component dependencies enhances component adaptability, maintainability, and reusability. More specifically, collaborating components are no longer aware of one another's interfaces, rather they are only aware of the events which they are interested in receiving. The

dependency, then, is not upon operation signatures, but upon an agreed event model that is operation signature independent. It becomes possible, therefore, to change operation signatures, and to even replace components with radically different alternates, providing the agreed event model is respected. This leads then to a decoupling not just in terms of component interfaces, but even in terms of component existence. Since components are no longer "pulling" events from one another, they no longer need reference to one another's identity. Relying on event "push" through a broker implies that a given event-pushing component could readily be replaced with multiple components, each pushing a (potentially overlapping) subset of the events of the original. Even removing a component completely, where that component was the sole generator of certain events, will have only one side effect: the events it generated will no longer be received by anyone.

Impact on Legacy Systems: Implicit Invocation decouples components; this increases their potential for independent evolution, substitution, or even elimination in the light of requirements change. This reduces system petrification, since changes are now more likely to be localisable within component boundaries. Implicit Invocation should also alleviate human petrification, since developers will be less fearful of unexpected side effects resulting from changes impinging on intricate inter-component dependencies.

Known Exceptions: Implicit Invocation is extremely effective at decoupling components. Experience has taught us, however, that it should not be applied indiscriminately. Rather, we have found it beneficial mostly for "high level" interactions between essentially distinct "large grained" components. At such high levels of abstraction, large grained components can be thought of as substantial black boxes, and the (newly decoupled) interactions among them often have some direct traceability to concepts that appear in the business requirements. Appealing to Implicit Invocation for smaller grained ("internal") components, however, can have a very negative impact (besides the likely performance decrease) on system comprehension. Excessive appeal to Implicit Invocation can result in an unmanageable proliferation of event interest registration and broadcast. Although the system can probably cope with this, we have found it more difficult for humans to do so. Small-grained Implicit Invocation tends to result in an overly "bitty" architecture, wherein there is insufficient cohesion amongst conceptually closely related parts of the system for people to "see the big picture". Appeal to traditional explicit invocation *within* a large grained component, then, provides an intra-component cohesion that aids the developer in grasping the behaviour of that component as a whole.

Example: Many of EDP's applications refer to product catalogues, which organise products for sale across a user-defined hierarchy of product categories. For example, "Golden Delicious" may be in the category "Apples", itself a subcategory of "Fruit". Now, if all Apples were put on special promotion, they may also need to be placed into a (possibly new) category of "Special Offers". Until recently, EDP's applications kept track of such category changes via continual reference to a database - which is both slow, and pessimistic (since usually no changes will have occurred). To circumvent such overhead, various caching schemes have been attempted at various times wherein database updates all pass through an interface which tracks recent changes. Components then no longer interrogated the database directly, but rather continually invoked functions in the caching component explicitly to check for changes. Unfortunately, such schemes quickly became overly convoluted, since, to prevent unacceptable time lags, components that needed to know about catalogue changes ended up needing to be "nudged" into invoking change checks on the cache as soon as changes occurred. This eventually lead to an intricate network of rules about who to "nudge" when, with each component usually having different means for performing that "nudging". Such approaches have proven to be an eventual maintenance nightmare, since they march towards tighter and tighter coupling between components in ever increasing attempts to overcome performance problems and delayed awareness of changes to the product catalogue. Consequently, EDP has now completely scrapped this approach, and has migrated to an Implicit Invocation scheme, wherein components register their interest in catalogue changes with a shared broker. This broker is notified as soon as the catalogue is changed, and immediately notifies all (and only) the components that have registered an interest in such changes. Appeal to Implicit Invocation has simultaneously simplified component code (significantly loosening component coupling), increased performance (since components are no longer pessimistically polling the database), and eliminated the time lag between catalogue changes and their recognition (notification) in interested components.

Alternates:

1. Wrappers

In a sense, the brokers underpinning Implicit Invocation are a sophisticated form of Proxy [Gamma et al. 1995]. These proxies "stand in for" the components that interact with them, shielding components from changes to each other’s implementation. We do not have to appeal to Implicit Invocation to receive all the benefits that proxies bring.

We can, for example, commit to unwavering interfaces each implemented permanently by a proxy component, which acts as a Facade [Gamma et al. 1995] to other "hidden" components. Such proxy facades are often termed wrappers. The components hidden behind a wrapper may then be modified quite radically in terms of their interface, the implementation, and even their number, with only the wrapper internals needing to change, thus shielding other components from any side effects resulting from the encapsulated modifications. Wrappers have great utility in decoupling components from one another's interfaces and implementations. Unlike Implicit Invocation, however, they are not as good at shielding us from the complete elimination of a component, since they perpetuate the "pull" model rather than the "push" model, and thus cannot prevent a operation being called. In such circumstances, we can appeal to the Null Object pattern [Woolf, 1998], with which to provide default behaviour behind a wrapper should some anticipated component be absent.

B.9. It's Not My Concern

Name: It's Not My Concern

Type: Productive

Summary: Push persistence code out of the main application and into the persistence service itself, thus disentangling and separating interaction with persistent storage from business related code, increasing understandability and maintainability of both concerns.

Target: Persistent Problems (p.203)

Features: Persistence is a separate concern from the main business-process-oriented concern of an application. Those concerns can be separated from each other by pushing code relating to persistence out of the application and into the persistence service. Modern day persistence services (in particular, DBMSs such as Oracle and Informix) provide sophisticated facilities for indexing, triggers, stored procedures, and so on. The facilities allow much persistence-related code to be pushed out of the application and into the persistence service itself. At the very lowest levels, translation between persistent and in-memory data formats for base data types, and their Composites [Gamma et al. 1995], which are not intrinsic to the DBMS can be handled by Serializers [Riehle et al. 1998] in the persistence service. As a higher-level example, code that manages dependencies between related updates can be pushed into persistence service triggers (are procedures invoked whenever specified events occur on persistent store). Likewise, code that performs computation on persistent data can be pushed into stored procedures, which are invoked by the persistent store whenever mentioned in queries, updates, or other manipulations. As a final example, performance concerns when managing large amounts of data are often mitigated by complex code providing clever shortcuts to related data. Embracing persistent storage indexes, which realise shortcut access automatically, can eliminate this code.

Benefits: Pushing behaviour out into the persistence service simplifies application logic by ensuring that the main body of the application focuses upon the concern of supporting business needs that can be traced back to requirements. This focus on supporting business needs enhances understandability and maintainability of the application. In addition, appeal to facilities provided by the persistence service ensures the consistent application of rules expressed therein across the whole application rather than relying upon careful respect for those rules in explicit code throughout the

application. For example, mandating that when a customer reaches their credit limit no more orders may be placed can be enforced by a trigger, which cannot be circumvented by application logic. Finally, appeal to facilities provided by persistence services can improve performance in many cases, since the algorithms therein have typically been fine-tuned over many years for their specific purpose.

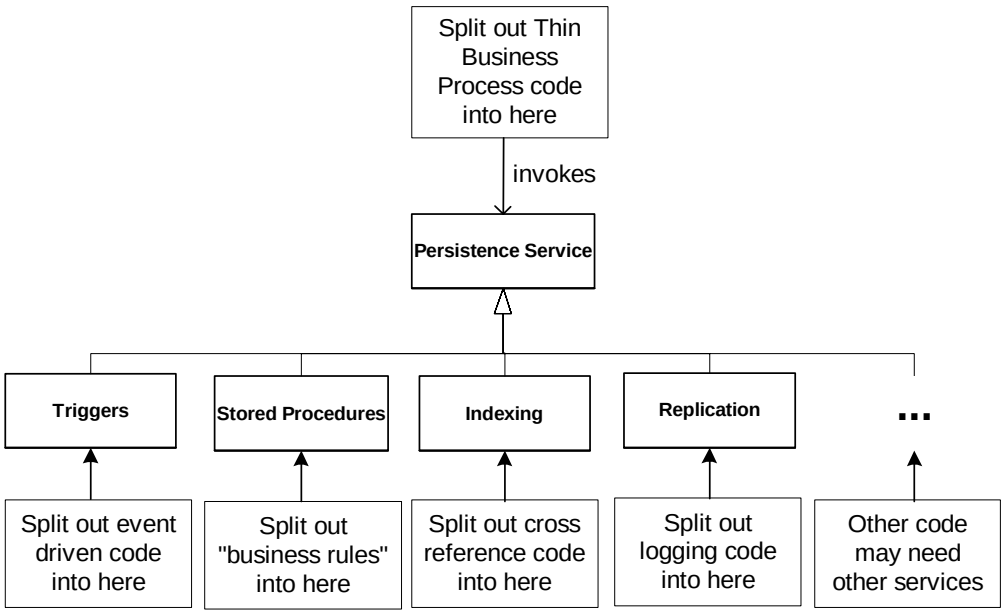

Figure B.9.1 – It's Not My Concern

Impact on Legacy Systems: Separating out business-oriented code from code managing persistence rules enhances the maintainability of each. Furthermore, the resultant traceability between business-oriented code and the application requirements enhances the potential for business-code to keep pace with changes in those requirements. This reduction in the lag between requirements change and reflection in supporting code contributes to the reintroduction of adaptability into a petrified legacy system.

Known Exceptions: In some applications, persistent code is essentially copying byte streams directly in and out of raw storage, perhaps via memory mapped files. These operations are often too low level to abstract out into useful stored procedures, triggers, index-assisted queries, etc. Of course, even low level persistence operations will benefit from a library of operations at a (even slightly) higher level of abstraction. A second exception is where application logic is not particularly obscured by persistence code, and hence adding a level of indirection will not improve maintainability (although it may possibly still improve performance).

Example: To achieve the separation of persistence code from business code at EDP, we have appealed to a wide number of database services written by this author for a

subsidiary of EDP during an earlier project. In particular, application-transparent database services for indexing, distribution, triggers, fragmentation, cascading, stored procedures, and a virtual file system have all been embraced. To tackle the problems in the example described in the Persistent Problems petrifying pattern (p.203), we have taken the following steps: Firstly, code that tracked and logged changes to order records has been removed from the application and pushed into various types of database trigger (e.g., before- and after-write). The use of triggers has ensured consistent enforcement of persistence rules across the application. A second change is that electronic orders in Unix files are now accessed via the database's virtual file system. The virtual file system makes the physical storage mechanism transparent, and hence these orders can now be accessed through the standard database interface as if they were actually stored in the database. Attached read and write triggers enable transparent conversion from and to the foreign format of the Unix file. As a final example, the elimination of cross reference files from the application and their replacement with direct utilisation of indexing mechanisms provided by the database management system has simplified the application and increased performance considerably. In summary, the elimination of a great deal of persistence code from the application has reduced its complexity, increased consistency, improved performance, and enhanced the ability of application developers to comprehend and maintain that code in the light of ongoing requirements change.

B.10. Legitimate Architecture

Name: Legitimate Architecture

Type: Productive

Summary: To enable a clear mapping from business process change to corresponding changes in the supporting information system, we need to directly reflect support for those business processes. This means a shift away from just supporting low-level constituent activities across scattered code, towards supporting business processes more explicitly. This implies continually moving towards a Legitimate Architecture aligned with the abstractions inherent in and traceable to the supported business processes.

Target: Accidental Architecture (p.178)

Features: A Legitimate Architecture is an architecture that directly reflects the commonalities and variabilities inherent in business processes being supported. Our assumptions about architectural commonality and variability, then, need to be grounded in an understanding of those business processes. For this, we can appeal to domain analysis techniques [Prieto-Dias and Arango, 1991] to tease out the points of commonality that are invariant across the business processes being supported, and the variabilities that capture the points of divergence between elements of those business processes. The sources for that knowledge are many, for example the existing code itself, the experience of business people and developers, and also possibly external sources (see also Gold Mining, p.242). Building upon this domain understanding, we construct a Legitimate Architecture reflecting points of commonality and variability in the domain. The important lesson we have learned at EDP, though, is that domain analysis can only give a commonality and variability snapshot, covering current and anticipated future requirements. Unanticipated future requirements change has the potential to upset the commonality and variability assumptions upon which our Legitimate Architecture is based. Hence, domain understanding and system architecting are necessarily perpetual processes. In other words, rather than expecting to be entirely speculative (proactive architecting) we must also be attuned to and responsive to the directions in which unanticipated changes nudge us (reactive architecting). We need, then, to remain continually vigilant by monitoring the evolution of the business processes we are supporting (a process we term Productive Evolution), and reflect that evolution via a continual rearchitecting (usually via refactoring [Opdyke, 1992; Fowler, 1999b]) of the system’s architecture to track changes in points of commonality and

variability brought about by unanticipated requirements change. A substantial and evolving catalogue of Regression Tests ensures that we are doing no harm when making these changes.

Benefits: Quite simply, a Legitimate Architecture enables traceability between requirements and implementation. This assists developers in constructing consistent mental models both of the business processes being supported and the architecture of the system that gives that support. Changes to requirements should then be mappable to corresponding changes in system implementation. That is, developers can better understand the scope and potential side effects of a given change request and thus negotiate legitimate strategies with which to address those change requests, whether that be by localised changes at points of variability or a rearchitecting of the system to reflect new commonality and variability assumptions.

Impact on Legacy Systems: A Legitimate Architecture is the antithesis of a legacy system. If we are successfully tracking requirements change, the system is clearly not petrified. Of course, for a system that is already petrified we need to legitimise its currently petrified accidental architecture. We achieve this by the same principles of domain understanding and rearchitecting that underpin Productive Evolution, except that at first we are concerned not with future requirements but with the backlogged requirements that the petrified architecture already resists. We address prioritised requirements from this backlog, one by one (or sometimes in aggregate groups) in a process of Productive Migration. Each backlogged requirement reveals inappropriate commonality and variability assumptions in the current architecture that need to be rectified (usually via refactoring) before that backlogged requirement can be legitimately supported by the system. As the backlog diminishes, the system's architecture becomes progressively more legitimate in terms of its accurate reflection of the commonality and variability requirements of the business processes being supported. Gradually, we transition away from the diminishing backlog towards new incoming requirements, shifting into Productive Evolution of a permanently adaptable system underpinned by an ever-evolving Legitimate Architecture.

Known Exceptions: As we have seen, a Legitimate Architecture is essentially an architecture that supports the construction of mental models that capture traceability from requirements to implementation. In cases where a system's current architecture is currently well understood (perhaps by deeply immersed expert designers), this traceability may still be possible without a Legitimate Architecture. Hence, the effort

required to legitimise the architecture may be seen as better-invested elsewhere. Importantly, though, new requirements, staff changes, and assignment to other projects, can erode the lucidity of hard-earned mental models, and recapturing such models is likely to be extremely hard work. Of course, for very small systems it may not be worth legitimising the architecture, since recapturing a lost mental model is unlikely to be too difficult. Finally, it is vital that the rearchitecting is underpinned by a sound and ongoing domain analysis. Unless we have confidence in our domain understanding, we can have no confidence in the "legitimate" architecture emerging from it. Thus, unless we can perform an effective ongoing domain analysis there is little point in striving for a Legitimate Architecture.

Example: EDP has a host of overlapping applications each with varying levels of petrified accidental architecture. To address the consequent divided maintenance burden, management has now committed to unifying the products via a shared legitimised architectural *framework* [Fayad et al. 1999]. This is seen as a long-term direction, probably taking five years or more. For the last couple of years there has been a shift at EDP to refactoring these diverse products towards a common Legitimate Architecture as both backlogged and new requirements are addressed. Importantly, this refactoring effort is entirely requirements driven. Thus, the systems have continued and will continue to duplicate overlapping functionality while new or backlogged requirements stimulate a gradual erosion of the duplication and the emergence of a shared Legitimate Architecture. Points of variability within the shared architectural framework allow different products to remain, since they do address slightly different markets. That is, EDP is not aiming for a single unified product, but rather for a legitimised shared product line architecture [Weiss and Lai, 1999] for a family of related products. See also [Lauder and Kent, 2000c] for a description of this direction for EDP, and for an overview of how a shared product line architecture can be modelled and deployed.

B.11. Lipstick on the Pig

Name: Lipstick on the Pig

Type: Productive

Summary: Switching to a contemporary graphical user interface increases the longevity of an application, enhances application maintainability, and permits separate customisation of the user interface.

Target: Ugly Duckling (p.214)

Features: Lipstick on the Pig is replacement of a dumb terminal user interface with a GUI interface. But what if we wanted to put Lipstick on the Pig and found that the pig didn't have any lips? Firstly, we need to disentangle user interface code from the application logic (a classic client-server strategy), and then we can adapt the separated user interface to a Windows or Internet-Browser-based GUI giving a contemporary look to the application. Usually, when a switch to a GUI has occurred, the flow around the screen is no longer controlled (dictated) by application logic but by an interaction between the user interface code and the user, thus giving a contemporary feel to the application. At the very least, the painful reliance on remembered keystrokes for navigation around the application tends to be reduced in favour of a more intuitive point-and-click navigation system.

Benefits: Lipstick on the Pig is a form of User Interface Modernization [Comella-Dorda et al. 2000]. It gives a contemporary "look and feel" to an application, thus reducing user perception that the application is archaic. A new user interface can, therefore, increase the longevity of an aging application, giving breathing space while the underlying application is itself tackled. The disentangling of user-interface code from application logic results in application logic that is more readily comprehended and maintained, in preparation for this later tackling of the application. An additional benefit of separating the front-end interface from the application back-end is, of course, that front-end can be adapted independently of the back-end modifications thus permitting customisation of the front-end, or indeed the back-end, to the needs of particular users.

Impact on Legacy Systems: Decoupling application logic from user interface code increases maintainability and thus helps reduce petrification. However, this separation has proven at EDP to be extremely challenging, and the effort involved should not be underestimated.

Note that the name "Lipstick on the Pig" derives from the observation that even though the application now looks pretty on the surface, the makeover is essentially an illusion. To those in the know, the application still remains a pig beneath all the makeup. To tackle the pig underneath, other suitable productive patterns would need to be applied. Nevertheless, Lipstick on the Pig has the tremendous benefit that it satisfies a major and pressing user demand for a contemporary user interface without mandating a complete rewrite of the whole application. This can certainly extend the life of a legacy system, thus generating a revenue stream with which to fund the deployment of further productive patterns behind the scenes.

Known Exceptions: Obviously, without adequate display hardware, it is impossible to provide an effective GUI interface. Although a windowing system can be partly simulated on some dumb terminals, the lack of a mouse means that such systems are still operated by magical key sequences. In addition, some highly experienced users have told us that a GUI slows them down. We need, therefore, to consider the needs of, and the impact upon, the users themselves when changing user interface strategies.

Example: At EDP a number of experiments were undertaken to determine a suitable implementation strategy for Lipstick on the Pig. Initially, user interfaces were developed using Java and Swing. This combination offered great portability with the promise of an ability to switch look-and-feel across a variety of native user interface emulations at runtime. Java and Swing certainly delivered in terms of being portable but suffered from two demerits in the process. Firstly, the execution speed was unacceptably slow; there were noticeable and uncomfortable delays while Swing responded to user interaction with the interface. Secondly, the user interface was only an approximation of the native interfaces it emulated. For example, Drag-and-Drop did not work well, and this was an important feature for EDP. Consequently, the Java/Swing approach was dropped, and the decision was made to sacrifice portability and adopt Microsoft Windows-specific user interfaces using Visual C++ with MFC. The resulting applications executed extremely quickly, and the user interface was (not surprisingly) intuitive and familiar to the experienced Windows user. As a result, EDP has committed to developing platform-specific user interfaces where possible. The lesson learned here was that portability of user interfaces has a price, in terms of performance and functionality, and that wherever possible portability should be sacrificed in favour of user satisfaction with their interaction with the application.

As another example, for applications that need to work over the Internet, EDP has developed customisable browser-based front-end lipstick using Dynamic HTML[Goodman, 1998], talking to a back-end Façade [Gamma et al. 1995] (with versions written in Java, C++, and Perl) via Java Server Pages [Bergsten, 2001].

Alternates:

1. You've Got The Look

To address the Ugly Duckling, we need to make a early decision with respect to whether or not we will undertake the separation of user interface code from application logic. One development group at EDP does not wish to do this, since they believe that the undertaking is risky. Consequently, that group will continue to write user interface code, albeit graphical, within the application itself. This approach has the advantage that change is minimised, but only addresses the user interface "look" (now graphical, rather than dumb terminal) without addressing user interface "feel". Since the flow of user interface code is intermingled with the flow of application logic, the user's interaction with the interface must follow this flow rather than following the flow most natural to the user. A number of experienced Windows users have found this controlled flow of EDP's GUI-ised applications rather stifling. However, for those who wish to minimise code changes, this option may remain attractive. Note, though, that in-lining GUI code perpetuates the intertwining with application logic that complicates program flow and impedes comprehension and maintainability. Consequently, this approach may impede the resolution of petrification in the modified legacy system. The majority of EDP legacy systems are undergoing a gradual but more radical user interface overhaul than this in-line GUI code strategy, and are pursuing a full-blown deployment of Lipstick on the Pig.

B.12. Negotiated Goal Alignment

Name: Negotiated Goal Alignment

Type: Productive

Summary: Inject drive and productivity into a flagging project, and make that project rewarding to team members, by negotiating an alignment of each individual's own goals with the goals of their role in the project.

Target: Black Hole (p.184)

Features: To inject productivity into flagging long-term projects (i.e. Black Holes), reward staff based upon the business contribution of the project upon which they work. This, however, is hard to achieve in practice - since speculative projects on the leading edge may offer significant potential value to the business but offer no immediate value during their development. Instead of relying upon monetary reward alone, strive to make the projects exciting and thus enticing to ambitious candidates. The key to this, we have found, is Negotiated Goal Alignment. In our experience, goal alignment seems to be a central contributor to a cohesive and motivated team. Find out what the team members personal and professional goals are, and negotiate an alignment between their goals and the needs of the project.

Benefits: Making a project enticing will attract new blood, with fresh insights and without the emotional baggage that goes with a long association with a failing project. Focusing on mutual goal alignment injects drive into that project, enhancing staff motivation and thus productivity. When productivity is high, attention is focused upon getting the project out of its rut, and instead delivering real business value. Once a team is productive, we will, of course, want to protect that productivity, and can appeal to other patterns with precisely that purpose [Taylor, 2000].

Impact on Legacy Systems: Legacy systems have petrified architectures. To add business value productively, we have to attack that petrification. This needs to become the focus of people working on a legacy system Black Hole. We need to align the goal of eliminating petrification with the goals of the project staff. If, for example, a staff member wishes to learn new technologies, they can focus upon new technologies often associated with migration. Likewise, a developer interested in improving software quality could be involved with writing Regression Tests to ensure that the replacement maintains the functionality of the original legacy systems. As a final example, somebody who simply wishes to escape from the project can be encouraged to learn

new skills during migration, which will better prepare him or her for other projects. As petrification is gradually eliminated, the project should climb out of its Black Hole, and back into productivity. Thus, the project will progressively become associated with success rather than failure. A successful project has a better chance of retaining and attracting good people.

Known Exceptions: There are, unfortunately, some projects that are simply so unpleasant that it is difficult to see how they could be dragged out of the doldrums. For example, a project may be so large and complicated that it literally will drag on for years, no matter what level of enthusiasm is injected. These are "Grin and Bare It" projects, where there seems little alternative but to plough on.

Example: There is a rather large project at EDP which is central to the business yet is a Black Hole, which even new staff specifically hired for the project subsequently refuse to work on. This particular project, like many Black Holes, revolves around maintenance of a legacy system in urgent need of migration to an adaptable replacement. After many years "in the doldrums", fresh enthusiasm has now been injected into this project by focusing staff involvement not upon the project itself, but rather upon both the leading-edge technologies involved in migration, and in the whole process of Productive Migration. Essentially what EDP is now saying to developers is "You are right, this legacy system is a dead end. Instead of just maintaining it, lets rescue it. We need your help to save its life." This approach has, to our great satisfaction, made the legacy system migration project the hottest project among developers at EDP. They see a threefold benefit: they learn about the core application without being dragged into a Black Hole, they gain career-enhancing skills in exciting migration technologies, and they are perceived as heroic saviours from a major impediment to the business.

Alternates:

1. Hire Mercenaries

A company in trouble may choose to hire mercenary developers, who will do any job no matter how dirty, but at a price. Mercenary developers are often highly skilled, thick-skinned, "can do" guns for hire. Mercenaries can be brought in to bite the bullet, and tackle a Black Hole once and for all. This approach can work if the mercenaries are paid according to the business value they deliver. Unfortunately, in our experience, many mercenaries work for bandit Consultancy Firms whose main objective is to keep fees high, hours long and goals vague. Without careful management by the client,

mercenaries can perpetuate a Black Hole project, making it not just a sinkhole for energy and time, but also for large amounts of money. Even when mercenaries do deliver, there can be problems when they leave if the client's own staff members were not working with them side by side. If mercenaries leave, taking all project expertise with them, there may be a new Black Hole awaiting the unwary.

2. Give Up

There may be projects that are so deeply in trouble, it makes more sense to simply scrap them and pursue an alternate route.

3. Blood Money

Although money is not the only motivator, it has been said that everybody has his or her price. Even deeply unmotivated staff can often been made to salivate when a big cheque is dangled in front of them. It is important, though, to reward people for the business value they deliver, rather than the overtime hours they work. The former encourages productivity, whereas the latter encourages heel dragging. We have certainly seen at EDP that a flagging project can be reinvigorated by the promise of a bag of gold at the end. We have also seen, though, that the promise must be kept, since failure to live up to the promise can leave resentful staff even more de-motivated for future projects.

B.13. Past Masters

Name: Past Masters

Type: Productive

Summary: Capture experiences, both good and bad, in an evolving catalogue of patterns to guide the work of those that follow, steering them away from trouble and towards successful practices.

Target: Trial and Error (p.212)

Features: To learn from the past, we must capture experience and share it. This includes both good experiences and bad ones. Consequently, this whole project has advocated the capture of lessons learned in terms of a pattern catalogue. Patterns capture experience that inspires subsequent discovery and invention. Bad experiences are expressed in terms of contra-indicated patterns best avoided, and good experiences are expressed as curative antidotes. It is imperative that pattern catalogues are cultivated according to the evolving needs and experiences of the organisation using them, to ensure their relevance to the specific systems that organisation maintains. To realise a pattern-oriented culture, however, can be extremely challenging. For example, many organisations reward gurus whose knowledge is in short supply. Sharing that knowledge can undermine a guru's relative worth. There must also be in{Kotter 1996 ID: 174} place a reward system that encourages, or at the very least does not penalise, sharing of prized knowledge. Unfortunately, introducing a patterns-oriented culture [DeLano and Rising, 1996; DeLano and Rising, 1997], as with any major cultural change [Kotter, 1996], requires considerable delicacy, energy, and upper management support. Leading such change appears to require quite considerable talents [Buchanan and Boddy, 1992], and hence success will not come easily.

Benefits: By following a development process that is driven by the cultivation of and appeal to evolving pattern catalogues, developers can learn from the past how to avoid a cycle of poor practices which have already been seen to be problematic and embrace practices which have already proven beneficial.

Impact on Legacy Systems: Appealing to catalogues of contra-indicated patterns underpinning legacy system petrification can highlight the current problems present in specific legacy systems, and a catalogue of corresponding antidote patterns helps guide the migration of those legacy systems back to adaptable systems. In essence, this whole project is underpinned by such an approach.

Known Exceptions: Learning from experience can never be a bad thing, unless of course we follow a “false prophet”. Many idiomatic practices at EDP proliferate the code yet are grounded in unsubstantiated folklore. For example, early code restricts variable names to six letters based upon the completely unfounded belief that this results in faster object code than if longer names were used. Another example is the avoidance of certain language features (e.g. templates in C++) because an experienced developer had a "bad" experience with them years ago. Consequently, convoluted idiomatic practices come and go without explanation and their justifications are lost in the history of development. Such practices, which we sometimes call the *Hocus Pocus* petrifying pattern [Lauder and Kent, 2001a], have lead to code that is unnecessarily hard to understand, and an unwillingness among developers to make changes to such code in case earlier generations "knew something we don't". Old habits die hard, but we have succeeded in breaking some bad habits at EDP by encouraged developers to annotate idiomatic code with textual justifications rationalising the idioms with reference to specific benefits. We also need to remember that a pattern catalogue is only a vehicle; it is not experience itself. Pattern catalogues can help transfer knowledge, but there is no substitute for having a willing mentor to hand. Even where pattern catalogues are available, it still requires considerable maturity of judgement to understand when to apply a given pattern and when to leave it well alone.

Example: The whole research project from which this pattern catalogue has been cultivated is underpinned by appeal to Past Masters. At EDP, then, we have looked at previous and emerging experience to capture patterns of what causes petrification and how to reverse and steer clear of petrifying practices in future.

Alternates:

1. We Only Employ the Best

Many companies claim to "only employ the best", and in their recruitment campaigns demand that "only the best need apply". In the context of software development "the best" are probably highly experienced, highly creative, certainly working at the highest level of developer maturity, and able to tackle extreme challenges effectively without reference to explicit pattern catalogues. In effect, "the best" can often find effective patterns for themselves, or have already learned "the tricks of the trade" elsewhere. Such people already have rich pattern catalogues in their heads, and employ them and cultivate them personally. Unfortunately, in reality there are only so many of "the best" to go around. As Ralph Johnson has said "On average, the average company employs

average people". Most organisations, then, will have only a small portion of "the best" and will need also to work with "the rest". Mentoring can help here, and a shared pattern catalogue is a powerful tool for doing just that.

B.14. Systematic Reuse

Name: Systematic Reuse

Type: Productive

Summary: Systematically cultivating and deploying reusable assets has the potential to shrink product size, enhance product quality, and boost productivity in the long run.

Target: Reuse Abuse (p.205)

Features: All artefacts across the software development process have the potential to be reusable. The Catalysis methodology [D'Souza and Wills, 1998] offers valuable insights on the reuse, and more importantly the refinement, of many development artefacts. In addition to technical concerns, [Reifer, 1997] and [Jacobson et al. 1997] address human factors that are central to reuse success.

Perhaps the most basic form of artefact reuse is coincidental reuse, which is essentially opportunistic and is characterised by periodic scavenging for the coincidental existence of a current artefact that just happens to suit a new context. Systematic Reuse, on the other hand, is a deliberate process of cultivating artefacts with the specific intent of them being reusable.

The result of a Systematic Reuse process is not artefacts that by chance happen to be reusable, but a continually evolving collection of cultivated reusable assets. We use the term asset to denote that these artefacts are known to have a reuse value. We use the term cultivated to indicate that they emerge over time with that reuse value in mind. This is important; although we can do a lot to enhance the desirability of a reusable artefact, such as making it simple to understand and deploy [Foote and Yoder, 1998], it will only be truly reusable if it meets our actual domain needs. Thus, reusable artefacts can rarely be planned up-front, rather they need to be allowed to emerge as our understanding of system requirements increases over time [Foote and Opdyke, 1995]. This implies a perpetual process of domain analysis throughout the life of our products, and a continual refining of our reusable assets to reflect shifting commonality and variability across that domain. We have found, however, that the asset value of a reusable artefact is volatile as its fitness for reuse fluctuates with ongoing requirements change A Systematic Reuse culture, then, must purposefully retain the reuse value of such assets by a deliberate process of ongoing cultivation and evolution in the light of requirements change.

Benefits: When Systematic Reuse is in place, application code size and other artefacts will shrink and development productivity should increase. Furthermore, the delivered products should be of higher quality, since they employ (reuse) previously debugged and field-proven assets.

Impact on Legacy Systems: Reuse is about recognising commonality and variability. Legacy systems exist where we have circumvented commonality and variability. To tackle petrification we need to legitimise the system's architecture, and this mandates a re-evaluation of our commonality and variability assumptions and their reflection in reusable assets. The benefits that reuse brings (see above) should alleviate the maintenance burden of a petrified legacy system. Perhaps more importantly, through, Systematic Reuse stimulates us to think continually about our commonality and variability needs, and how they are best reflected in legitimate reusable assets, thus steering us away from the hacking practices that underlie system petrification.

Known Exceptions: The introduction of a Systematic Reuse culture is, of course, a potentially major business process change, involving multiple socio-technical challenges, requiring both genuine management buy-in [Reifer, 1997] and appeal to an ongoing and sanctioned change-management program [Jacobson et al. 1997]. For cultures that resist such change, and where we cannot engender true support for reuse from on high, rather than mere lip service, then a reuse program is likely to be ineffective.

Also, like any business process change, the shift to a Systematic Reuse culture only makes sense if it can demonstrably contribute to satisfying business goals. In cases where business goals cannot accommodate long-term amortisation of the costs inherent in Systematic Reuse, it may be more appropriate to rely on opportunistic or small-scale reuse and operate reuse on a localised basis.

Note that reusable assets are frequently much more expensive to develop initially than are ad-hoc artefacts. The costs of developing a reusable asset can only be justified when we know that it will actually be reused. To develop a reusable asset purely speculatively, with no concrete need demonstrable, risks time, money, and effort, all of which may be better placed elsewhere. In particular, it is difficult to know exactly what is needed of a reusable asset until multiple concrete instances have been shown to require it.

It could be said that "one is a fluke, two is coincidence, three is a trend", and this appears to be good advice for the development of reusable assets. That is, there should

probably be at least three demonstrable concrete contexts that would benefit from the existence of a common artefact, and from which the requirements for that artefact can be gleaned and costs amortised.

Example: At EDP we are (slowly) adopting a reuse program maintaining executable components [Szyperski, 1998], configurable component generation [Czarnecki, 1998], and (to a lesser degree) higher-level analysis and design reuse. The general direction at EDP is a shift away from the development of individual systems as separate projects, towards the production of software product line families sharing cultivated, focused, reusable assets [Coplien, 1998b; Bass et al. 1998; Bass et al. 1999; Weiss and Lai, 1999]. In particular, EDP is currently working towards application families constructed from reusable application frameworks [Pree, 1994; Fayad et al. 1999] covering the domains addressed by historically separate yet overlapping software systems [Lauder and Kent, 2000c]. Early results look promising, and at the time of writing, applications developed in this way are just starting to be delivered into customer hands. Meetings with EDP management have revealed that they recognise this to be a long-term undertaking, and are prepared for this strategy to take at least five years to be fully in effect throughout the company.

Alternates:

1. COTS

Every since the early promise of Software-ICs [Cox, 1984], EDP has been seduced by Commercial Off the Shelf (COTS) reusable components. Management at EDP has frequently asked developers to seek out reusable artefacts from elsewhere, instead to cultivating them in-house. Several “reusable” artefacts were purchased, primarily software libraries and components, but also designs in book form (e.g. [Fowler, 1997; Hay, 1996]). Unfortunately, EDP generally found that either the purchased products were of lower quality than could have been developed in-house, or required so much adaptation to suit EDP’s requirements that they were almost totally rewritten. Thus, despite EDP’s initial enthusiasm for COTS components, the general feeling now at the company is that reusable assets need to be cultivated from within, in order for them to continually and accurately reflect the evolving commonality and variability requirements of the domain in which those assets are to be reused.

B.15. Virtual Componentisation

Name: Virtual Componentisation

Type: Productive

Summary: First decide what components would ideally exist in the monolithic system, then implement a Façade [Gamma et al. 1995] on top of the existing system to provide the illusion of these components already being in place. Over time, real components can be substituted for the virtual components without mandating changes to the system's interface.

Target: Monolithicity (p.201)

Features: The ideal solution to the Monolithicity problem is, of course, to utilise manageable focused software components. However, switching to a completely component-oriented architecture in a single step is both expensive, in terms of time and money, and risky in terms of scale. A more cautious approach separates two concerns: the interface and the implementation. We can adopt, then, a two-phase strategy: the first phase focuses upon determining the set of components that would ideally exist, establishing component boundaries, determining their interfaces, and creating a set of corresponding virtual component interfaces on top of the existing legacy system. A virtual component interface is generally implemented as a Façade [Gamma et al. 1995] on top of a legacy system.

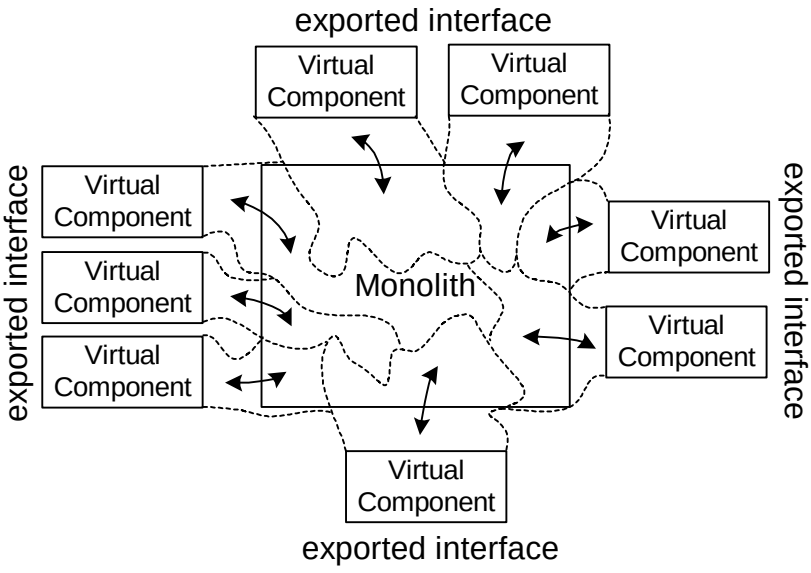

Figure B.15.1 – Virtual Componentisation

A virtual component provides the illusion of an actual component already being in place. Over time (the second phase), real components can be substituted for the virtual

components without mandating changes to the virtual interfaces. A mix of real and virtual components can be connected via a Mediator [Gamma et al. 1995], which encapsulates knowledge of their collaboration. One example of such mediation is the Gateway proposed in [Brodie and Stonebraker, 1995].

Benefits: Virtual Componentisation essentially stabilises interfaces by committing to them up-front in the virtual components. Thus, developers of code that invokes the currently monolithic system can adapt their own code to reflect those established interfaces safe in the knowledge that they will remain relatively stable, no matter what implementation eventually lies behind them as real components are put in place.

Impact on Legacy Systems: Virtual Componentisation can be implemented more readily that true componentisation since it requires only a shallow up-front understanding [Weiderman et al. 1997] of the (implicit) interface in the legacy system rather than an immediate deep reverse engineering. It can temporarily prolong the lifespan of a monolithic legacy system by giving the impression to the outside world that the system is already composed of components. This gives the maintainers of the legacy system a breathing space, relieving some of the time pressure upon them to sort out the internals of the system. Consequently they can take a more measured approach to their work "behind the scenes" breaking the monolith up into real components that adhere to the interfaces established by the virtual components. Developers can thus perform that real componentisation work incrementally, enabling them to interleave this work with other higher priority activities as dictated by evolving customer needs.

For those virtual components where real components are put in place, the legacy system will benefit from all the advantages that component based systems bring: maintainability, reusability, substitutability, and so on (see, e.g., [Szyperski, 1998]), all of which help reduce system petrification. However, for some virtual components, the benefits of putting real components in place over time may prove insufficient to warrant the effort, and thus, for them, the virtual component interface can remain a permanent wrapper around the legacy code. Indeed, there may even be the potential to scavenge the real components from out of the legacy system itself, although it appears that both estimating a legacy system's potential for component scavenging [Burd and Munro, 1997a] and then successfully scavenging components from that legacy system can be rather difficult in practice (see, e.g., [Burd and Munro, 1997b; Burd and Munro, 1998]).

Known Exceptions: It is difficult to think of any instance where Virtual Componentisation is undesirable, other than either for extremely small systems (which

can be thought of as a single component already) and also for cases where immediate dismissal is an option (see Alternates below).

Example: At EDP we are striving to realise Virtual Componentisation of the company's largest legacy system, which is a distribution management system. The sheer size of this system makes Virtual Componentisation a long-term undertaking. Thus far, we have succeeded in deploying virtual components for the most critical parts of the system: product cataloguing, pricing, order tracking, customer account management, and a number of related support areas. Results have been pleasing, and these virtual components have already been deployed "in the field" in the latest customer release for a number of “early adopters”. The next step, which is currently underway, is to break the parts of the system covered by the virtual components into real components "under the covers". EDP management recognises this as best tackled incrementally; and thus a realistic ambition for complete componentisation within five years has been set.

Alternates:

1. Immediate Dismissal

One obvious alternative is to replace the current monolith immediately with a collection of functionally equivalent components, and assume their interfaces as those to which callers must adhere. In our experience, however, it is unusual for components to be available up-front which are sufficiently similar in function to the existing system to enable their direct substitution without radical changes both in the code which invokes them and in the components themselves. We may, of course, choose to adopt a mixed approach, wherein we appeal to Virtual Componentisation, and for those areas in which substitutable components are already available we undertake such substitution immediately. The remainder of the system is then migrated to real components at a more leisurely pace.

B.16. Virtual Platform

Name: Virtual Platform

Type: Productive

Summary: Encapsulate calls to underlying facilities in a Virtual Platform, which provides a platform independent interface to those services, thus enhancing application portability now and in the future.

Target: Ball and Chain (p.181)

Features: To break ties with a specific platform, construct a portable fundamental services layer constituting a Virtual Platform, which establishes a platform independent interface to underlying facilities. Application code should then no longer make any direct calls to the underlying operating system, DBMS, or other services provider, but rather make all calls through the Virtual Platform. The Virtual Platform then encapsulates appropriate conditional code, which translates calls to its platform-independent interface into corresponding calls across the range of supported underlying real platforms.

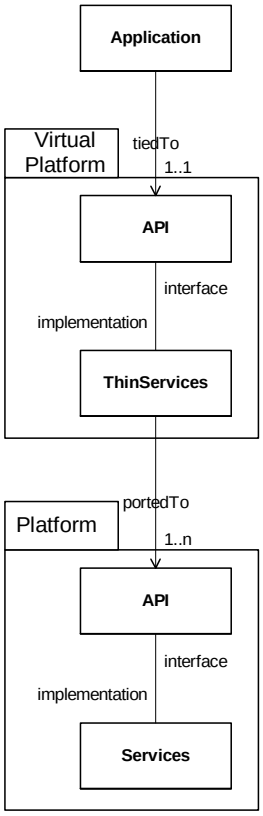

Figure B.16.1 – Virtual Platform

Where a required platform does not support a feature required by the application, that functionality is added to the Virtual Platform (either directly, or using an Extension Object [Gamma, 1998]) or, where that is impossible, the application's dependability upon that non-portable feature is eliminated.

Benefits: A Virtual Platform brings portability. Porting the Virtual Platform to new underlying platforms, then, brings with it portability of applications without requiring any code changes to the applications themselves. Portability concerns are now isolated to the Virtual Platform, freeing application developers to focus upon application-oriented issues. All applications built on top of a Virtual Platform benefit from increased portability when additional platforms are accommodated, thus a Virtual Platform amortises the cost of portability. A Virtual Platform is a form of insurance plan that ensures that the application's life will not be cut short by market whims with respect to platform preferences.

Impact on Legacy Systems: A Virtual Platform injects into legacy systems adaptability in the light of platform changes. Migrating of the legacy system to new platforms, then, is no longer a concern, leaving the maintainers of the legacy system to concentrate on catching up with evolving customer requirements. Note, however, that most legacy systems already commit to one or more specific platforms, and rewriting them to match the interface of the Virtual Platform may prove daunting. In such cases, we could front a Virtual Platform with an Adaptor [Gamma et al. 1995] to mirror the platform interfaces that the legacy system is already using.

Known Exceptions: Although a Virtual Platform is seen as widely beneficial, there are occasions where it is not appropriate. Development and maintenance of, as well as migration and adherence to, a Virtual Platform is a far from trivial task. Virtual Platforms are very time consuming both technically and managerially. That effort may only be worthwhile for applications that will be widely ported. Where there is an ironclad commitment to one or maybe two specific platforms, Virtual Platform support may prove too expensive for the benefits it brings. Furthermore, there are some contexts for which a Virtual Platform detracts from the effectiveness of the application. One example of this is graphical user interfaces. Despite numerous attempts at EDP, no mechanism has been uncovered that successfully supports a portable GUI without impairing the native look and feel with which users will be familiar. Portable graphical user interfaces tend to result in lowest-common-denominator support, which users may find unacceptable. In this case, we have found no alternative other than maintaining

parallel sets of diverse user interface code, one for each platform. This lowest-common-denominator concern may well extend to other areas beyond user interfaces, although we did not come across any that were equally disconcerting.

Example: Virtual Platforms have alleviated many portability and maintenance headaches at EDP. We have almost eliminated the need for compile-time conditional code by embedding such code within the Virtual Platform itself. Bolstered by the efficacy of its current Virtual Platform, and to achieve wider portability than EDP's current Virtual Platform accommodates, a new Virtual Platform is being built primarily on top of Doug Schmidt's excellent ACE toolkit [Schmidt, 1999]. ACE is a highly ported, and highly portable, low-level fundamental services layer. It is both freely available (including all source code), and highly efficient. In fact, ACE introduces absolutely no overhead whatsoever on top of native operating system calls.

Embracing a Virtual Platform clearly brings many benefits. However, one noticeable problem at EDP has been the occasional tendency of developers to code around the Virtual Platform and talk to the underlying platform directly when they are "certain" that all current platforms support that call identically. Unfortunately, in such cases it has often transpired that such universally supported native calls were not as universally supported as originally believed. This can be particularly troublesome when porting to a new platform, which was previously considered unimportant and hence was initially excluded from considerations of the universality of support for a given call. Consequently, it is recommended that Virtual Platforms support an interface for every required platform call, even where that call has a common interface across all currently supported platforms. Code reviews have certainly helped to catch numerous instances of native coding, but the best approach identified at EDP has been to force people to do their own porting when code is proven non-portable and then to fold that work back into the Virtual Platform. The tedious and frustrating nature of this re-work, and the embarrassment at having been "caught out" in front of colleagues has meant that such deviance is rarely repeated.

Alternates: Several experiments were conducted at EDP to identify an appropriate solution to the Ball and Chain problem. To begin with, various approaches to enhance portability were brainstormed and enumerated. Next, pilot projects were undertaken with all but one of these approaches. As a result of these experiments, Virtual Platform stood out from the others as particularly satisfactory, nevertheless the alternative approaches outlined below may appeal in certain circumstances:

1. Multiple Versions

The obvious possibility of maintaining separate versions of each application, one for each platform, was rejected almost immediately by all parties at the brainstorming sessions, since previous experience had taught EDP "the hard way" that maintaining multiple versions of an application can result in overwhelming version control problems. In particular, keeping all of these versions in line with one another in the light of ongoing requirements change can become a development and support nightmare.

2. In-Line Conditional Compilation

A number of languages (e.g. C and C++) support pre-processors, which permit conditional code to be evaluated during compilation. For languages that do not support pre-processors, they are relatively easy to develop, and indeed the standard C++ CFRONT pre-processor is readily adaptable to a variety of languages. Such pre-processors accommodate compile time decisions with respect to which code to include for the platform targeted by the current compilation. In-line conditional compilation, then, involves wrapping conditional code around calls to platform-specific features wherein a check is made for each supported platform, and the associated function call specific to a platform is compiled when that platform is identified as the current compilation target.

This approach is certainly effective, but it has the unfortunate side effect that it complicates code with conditional statements and multiple associated in-line platform-specific calls. At EDP, the resulting source code was found to be more difficult to comprehend and maintain than the corresponding platform-specific versions. A more troublesome problem encountered at EDP, though, was that as requirements demanded support for new platforms, it was necessary to undertake an exhaustive, tedious, and somewhat error-prone process of re-examination of the whole application in search of all locations where new conditional code was needed for those platforms. In summary, portability is traded off here against maintainability, and this may be an unacceptable trade-off. Despite these reservations, in-line conditional code may well be a good stop-gap approach since, at least at EDP, this was found to be an effective means of identifying instances of platform-specific code in the first place. A subsequent phase, then, may adopt one of the approaches outlined below to substitute non-conditional portable code, thus eliminating the associated maintenance burden.

3. Platform Emulator

A number of EDP legacy systems were originally targeted directly at Unix platforms. A considerable number of existing and potential customers demanded ports to Windows NT. Rather than attempt a rewrite using the Win32 API, EDP purchased a product named Nutcracker [DataFocus, 2000], which provides an emulation of the Unix API on top of Win32.

Nutcracker at first seemed like an enticing solution to the porting problem. Consequently, a couple of developers were devoted full time to the anticipated minor tweaks needed for a Nutcracker port. Unfortunately, over time, a number of problems emerged. The first problem is that a number of required Unix services were not emulated by Nutcracker, and consequently accommodating them involved the type of immersion in the Win32 API that EDP had wanted to avoid. The second problem is that Nutcracker proved to be slow in terms of execution time for a number of API calls, in comparison to their equivalents on Unix. In some cases these performance issues made it necessary to code around Nutcracker and make Win32 calls directly. The third problem is that Nutcracker itself runs on only Win32, and thus will be of no help whatsoever in terms of porting to other platforms (such as Apple Macintosh) should the need arise. The fourth problem is that Nutcracker is a binary-only product - EDP was did not have access to source code. Consequently, EDP was completely dependent upon the responsiveness of the Nutcracker developers to bug reports, and emulation support for further Unix features, as they became necessary. EDP management was understandably uncomfortable with this dependency: "we don't want our nuts in anybody else's cracker". In light of all of these concerns, EDP has come to view Nutcracker and similarly proprietary platform emulators as only a stopgap solution to portability.

4. JRE

Java can be thought of as employing a Virtual Platform in its runtime environment. Java's high level of portability makes it a sensible choice for many new applications. Existing systems, not written in Java, though, cannot benefit from Java's portability advantages. Although a small number of other languages have been ported to the Java Runtime Environment (e.g. [JPython, 2001]), these have not proven particularly popular. Furthermore, even for new applications Java may not be the appropriate language of choice. For example, a Java-based database administration tool developed at EDP was deemed sluggish and quirky in its user interface, and was necessarily rewritten in Visual C++.

References

Abrial, J.-R., Schuman, S. and Meyer, B. (1980) A Specification Language. In: McNaughten, R. and McKeag, R., (Eds.) *On the Construction of Programs*, Cambridge University Press.

Alexander, C. (1979) *The Timeless Way of Building*, Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M. and Angel, S. (1975) *The Oregon Experiment*, Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M. and Angel, S. (1977) *A Pattern Language*, Oxford University Press.

Austin, J.L. (1962) *How To Do Things With Words*, Clarendon Press.

Bach, J. (1995) Enough About Process: What We Need Are Heroes. *IEEE Software* March

Bach, J. (1997) Good Enough Quality: Beyond the Buzzword. *IEEE Computer* August

Bach, J. (1998) Microdynamics of Process Evolution. *IEEE Computer* February

Bach, J. (1999a) *General Functionality and Stability Test Procedure*, http://www.satisfice.com.

Bach, J. (1999b) Risk and Requirements-Based Testing. *IEEE Computer* June

Bach, J. (1999c) What Software Reality is Really About. *IEEE Software* December

Bach, J. (2001) *What is Exploratory Testing? And How it Differs from Scripted Testing*, http://www.satisfice.com.

Bass, L., Campbell, G., Clements, P., Northrop, L. and Smith, D. (1999) Third Product Line Practice Workshop Report (CMU/SEI-99-TR-003). Software Engineering Institute, Carnegie Mellon University.

Bass, L., Chastek, G., Clements, P., Northrop, L., Smith, D. and Withey, J. (1998) Second Product Line Practice Workshop Report (CMU/SEI-98-TR-015). Software Engineering Institute, Carnegie Mellon University.

Beck, K. (2000) *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

Beck, K. and Fowler, M. (2001) *Planning Extreme Programming*, Addison-Wesley.

Beedle, M., Devos, M., Sharon, Y., Schwaber, K. and Sutherland, J. (2000) SCRUM: A Pattern Language for Hyperproductive Software Development. In: Harrison, N., Foote, B. and Rohnert, H., (Eds.) *Pattern Languages of Program Design 4*, Addison-Wesley.

Bennett, K. (1995) Legacy Systems: Coping With Success. *IEEE Software* January

Bennett, K.H. (1998) *Do Program Transformations Help Reverse Engineering?*, Proceedings of International Conference on Software Maintenance, IEEE.

Bennis, W.G. (1966) *Beyond Bureaucracy*, Wiley.

Bergey, J.K., Northrop, L. and Smith, D.B. (1997) Enterprise Framework for the Disciplined Evolution of Legacy Systems (CMU/SEI-97-TR-007). Software Engineering Institute, Carnegie Mellon University.

Bergey, J.K., Smith, D.B., Tilley, S.R., Weiderman, N.H. and Woods, S. (1999) Why Reengineering Projects Fail (CMU/SEI-99-TR-010). Software Engineering Institute, Carnegie Mellon University.

Bergsten, H. (2001) *Java Server Pages*, O'Reilly.

Binder, R.V. (1999) *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley.

Black, M. (1979) More about Metaphor. In: Ortony, A., (Ed.) *Metaphor and Thought*, Cambridge University Press.

Blau, P.M. (1987) Microprocess and Macrostructure. In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Booch, G. (1991) *Object Oriented Design with Applications*, Benjamin Cummings.

Brodie, M.L. and Stonebraker, M. (1995) *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*, Morgan Kaufman.

Brooks, F.P. (1987) No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* 20, 4

Brown, A., Morris, E. and Tilley, S.R. (1996) Assessing the Evolvability of a Legacy System (Draft White Paper). Software Engineering Institute, Carnegie Mellon University.

Brown, K. (1998) *Formal Methods and Patterns*, http://c2.com/wiki?FormalMethodsAndPatterns.

Brown, W.J., Malveau, R.C., McCormick III, H.W.S. and Mowbray, T.J. (1998) *AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis*, Wiley.

Brown, W.J., McCormick III, H.W.S. and Thomas, S.W. (1999) *AntiPatterns and Patterns in Software Configuration Management*, Wiley.

Brown, W.J., McCormick III, H.W.S. and Thomas, S.W. (2000) *AntiPatterns in Project Management*, Wiley.

Buchanan, D. and Boddy, D. (1992) *The Expertise of the Change Agent*, Prentice Hall.

Burd, E. and Munro, M. (1997a) *Assisting Human Understanding to Aid the Targeting of Necessary Reengineering Work*, Proceedings of 5th Working Conference on Reverse Engineering.

Burd, E. and Munro, M. (1997b) Enriching Program Comprehension for Software Reuse. In: Proceedings of International Workshop on Program Comprehension.

Burd, E. and Munro, M. (1998) *Investgating Component-Based Maintenance and the Effect of Software Evolution: a reengineering approach using data clustering*, Proceedings of International Conference on Software Maintenance, IEEE.

Burd, E., Munro, M. and Pakstiene, S. (2001) Initial Recommendations for Improving Maintenance Strategy. In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change, volume 2*, Springer-Verlag.

Burr, R.J.A. and Casselman, R.S. (1994) Timethread-Role Maps for Object-Oriented Design of Real-time and Distributed Systems. In: *Proceedings of OOPSLA 94*, ACM.

Burr, R.J.A. and Casselman, R.S. (1996) *Use CASE Maps for Object-Oriented Systems*, Prentice Hall.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley.

Cagnin, M.I., Penteado, R., Braga, R.T.V. and Masiero, P.C. (2000) *Reengineering using Design Patterns*, Proceedings of the Working Conference on Reverse Engineering, IEEE.

CASE (2001) *CASE tool index*, http://www.cs.queensu.ca/Software-Engineering/tools.html.

Champy, J. (1995) *Reengineering Management: The Madate for New Leadership*, Haper-Collins.

Chikofsky, E.J. and Cross II, J.H. (1990) Reverse Engineering and Design Recovery: A Taxonomy. *Software* (January): IEEE.

Chu, W.C., Lu, C.-W. and Shiu, C.P. (2000) Pattern-based software reengineering: a case study. *Journal of Software Maintenance: Research and Practice* (12): Wiley.

Clarke, S., Harrison, W., Ossher, H. and Tarr, P. (1999) Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In: *Proceedings of OOPSLA '99*,

Cockburn, A. (1999a) *A methodology per project*, http://members.aol.com/acockburn.

Cockburn, A. (1999b) *Characterizing People as Non-Linear, First-Order Components in Software Development*, http://members.aol.com/acockburn.

Cockburn, A. (2000a) *Just-In-Time Methodology Construction*, http://members.aol.com/acockburn.

Cockburn, A. (2000b) *Legacy System and Lightweight Methodologies*, Personal Communication.

Cockburn, A. (2000c) *Methodological Failure*, Personal Communication.

Cockburn, A. (2001a) *Crystal/Clear: A Human Powered Methodology for Small Teams*, Addison-Wesley.

Cockburn, A. (2001b) *Software Development as a Cooperative Game*, Addison-Wesley.

Coleman, J.S. (1987) Free Riders and Zealots. In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Comella-Dorda, S., Wallnau, K., Seacord, R.C. and Robert, J. (2000) A Survey of Legacy System Modernization Approaches (CMU/SEI-2000-TN-003). Software Engineering Institute, Carnegie Mellon University.

ControlChaos (2000a) *Controlled Chaos Software Development*, http://www.controlchaos.com.

ControlChaos (2000b) *Philosophies of Software Development*, http://www.controlchaos.com.

ControlChaos (2000c) *Scrum Process Overview*, http://www.controlchaos.com.

ControlChaos (2000d) *The "Flip"*, http://www.controlchaos.com.

ControlChaos (2000e) *The Philosophy of Scrum*, http://www.controlchaos.com.

Cook, K. (1987a) Emerson's Contributions to Social Exchange Theory. In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Cook, K. (1987b) *Social Exchange Theory* , SAGE Publications.

Coplien, J.O. (1998a) *Formal Methods and Patterns*, http://c2.com/wiki?FormalMethodsAndPatterns.

Coplien, J.O. (1998b) *Multi-Paradigm Design for C++*, Addison-Wesley.

Coplien, J.O. (1998c) Reflection on the Seven Habits of Successful Pattern Writers. *C++ Report* (January): SIGS.

Coplien, J.O. and Schmidt, D. (1995) Pattern Languages of Program Design. In: Addison-Wesley.

Cox, B. (1984) *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley.

Crosby, P.B. (1979) *Quality is Free*, McGraw-Hill.

Czarnecki, K. (1998) *Generative Programming (PhD thesis)*, Technische Universitat Ilmenau, Germany.

D'Souza, D. and Wills, A. (1998) *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley.

Dai, P., Farmer, R. and O'Callaghan, A. (1999) *Patterns for Change - Sample Patterns from the ADAPTOR Pattern Language*, Presented at EuroPLoP 99.

DataFocus (2000) *The NuTCRACKER Platform* , http://www.datafocus.com.

Date, C. and Darwen, H. (1997) *A Guide to the SQL Standard*, Addison-Wesley.

Davenport, T.H. (1993) *Process Innovations: Reengineering Work Through Information Technology*, Harvard Business School Press.

DeLano, D.E. and Rising, L. (1996) *Introducing Patterns into the Workplace*, http://www.agcs.com/patterns/oopsla/intro.h.

DeLano, D.E. and Rising, L. (1997) *Introducing Technology into the Workplace*, http://st-www.cs.uiuc.edu/|plop/plop97/Proceedings/delano.pdf.

DeMarco, T. (1978) *Structured Analysis and System Specification*, Yourden Press.

DeMarco, T. and Lister, T. (1999) *Peopleware: Productive Projects and Teams (2nd Edition)*, Dorset House Publishing.

Deming, W.E. (1986) *Out of Crisis: Quality, Productivity, and Competitive Position*, Cambridge University Press.

Dewer, R., Lloyd, A.D., Pooley, R. and Stevens, P. (1999) Identifying and communicating expertise in systems engineering: a patterns approach. *IEE Proceedings Software* 146 (3): IEE.

DiBona, C., Ockman, S. and Stone, M. (1999) *Open Sources: Voices from the Open Source Revolution*, O'Reilly.

Dietz, J.L.G. (1994) Modelling Business Processes For the Purpose of Redesign. In: *Proceedings of IFIP TC8 Open Conference on Business Process Redesign*, North Holland.

Dyson, P. and Anderson, B. (1998) State Patterns. In: Martin, R., Riehle, D. and Buschmann, F., (Eds.) *Pattern Languages of Program Design 3*, Addison-Wesley.

Eatock, J., Giaglis, G.M., Paul, R.J. and Serrano, A. (2000) The Implications of Information Technology Infrastructure Capabilities for Business Process Change Success. In: Henderson, P., (Ed.) *Systems Engineering For Business Process Change*, Springer-Verlag.

Eden, A., Hirschfield, Y. and Yehudai, A. (1998) *LePUS - a Declarative Pattern Specification Language*, Department of Computer Science, Tel Aviv University.

Edwards, J. and Millea, T. (2001) Cheating Death (Better Software Evolution). In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change, volume 2*, Springer-Verlag.

Emerson, R.M. (1987) Towards a Theory of Value in Social Exchange . In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Eva, M. (1995) *SSADM Version 4: A User's Guide*, McGraw Hill.

FAMOOS (1999) *FAMOOS Handbook*, http://www.iam.unibe.ch/~famoos/handbook.

Fayad, M.E., Schmidt, D. and Johnson, R. (1999) *Building Application Frameworks*, Wiley.

Foote, B. and Opdyke, W.F. (1995) Lifecycle and Refactoring Patterns that Support Evolution and Reuse. In: Coplien, J.O. and Schmidt, D., (Eds.) *Pattern Languages of Program Design*, Addison-Wesley.

Foote, B. and Yoder, J. (1996) Evolution, Architecture, and Metamorphosis. In: Vlissides, J., Coplien, J.O. and Kerth, N., (Eds.) *Pattern Languages of Program Design 2*, Addison-Wesley.

Foote, B. and Yoder, J. (1998) The Selfish Class. In: Martin, R., Rohnert, H. and Buschmann, F., (Eds.) *Pattern Languages of Program Design 3*,

Foote, B. and Yoder, J. (2000) Big Ball of Mud. In: Harrison, N., Foote, B. and Rohnert, H., (Eds.) *Pattern Languages of Program Design 4*, Addison-Wesley.

Fowler, M. (1997) *Analysis Patterns: Reusable Object Models*, Addison-Wesley.

Fowler, M. (1999a) *Code Smells and AntiPatterns*, Personal Electronic Communication.

Fowler, M. (1999b) *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

Fowler, M. (2000) *The New Methodology*, http://www.martinfowler.com/articles/newMethodology.html.

Friedman, D. (1987) Note on "Toward a Theory of Value in Social Exchange". In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Gallagher, K. (2001) *Keith Gallagher's Home Page*, http://www.cs.loyola.edu/~kbg.

Gamma, E. (1998) Extension Object. In: Martin, R., Riehle, D. and Buschmann, F., (Eds.) *Pattern Languages of Program Design 3*, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides.J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Gane, C. and Sarson, T. (1979) *Structured Systems Analysis and Design*, Prentice Hall.

Gold, N. (1998) *The Meaning of "Legacy Systems"*, Report: PR-SABA-01, Center for Software Maintenance, University of Durham.

Goldkuhl, G. (1996) Generic Business Frameworks and Action Modelling. In: *Proceedings of Language/Action Perspective '96*, Springer Verlag.

Goldkuhl, G. (1998a) The Six Phases of Business Processes - Communication and The Exchange of Value. In: *Proceedings of Twelfth Biennial ITS Conference*, Stockholm:

Goldkuhl, G., Lind, M. and Seigerroth, U. (1998b) *The Language Action Perspective on Communication Modelling: Proceedings of the Third International Workshop*, Jonkoping International Business School.

Goodman, D. (1998) *Dynamic HTML*, O'Reilly.

Graham, I., Henderson-Sellers, B. and Younessi, H. (1997) *The OPEN Process Specification*, Addison-Wesley.

Green, T.F. (1979) Learning without Metaphor. In: Ortony, A., (Ed.) *Metaphor and Thought*, Cambridge University Press.

Griswold, W.G. (1991) *Program Restructuring as an Aid to Software Maintenance (PhD thesis)*, University of Washington.

Griswold, W.G. (1996) *Just-in-Time Architecture: Planning Software in an Uncertain World*, Proceedings of the 2nd International Symposium of Sofware Architecture.

Habermas, J. (1984) *The Theory of Communicative Action*, Polity Press.

Hammer, M. (1990) Reengineering work: don't automate, obliterate. *Harvard Business Review, July/August*

Hammer, M. (1996) *Beyond Reengineering*, Haper-Collins Business.

Hammer, M. and Champy, J. (1993) *Reengineering The Corporation: A Manifesto for Business Revolution*, Nicholas Brealey.

Harel, D. and Politi, M. (1998) *Modeling Reactive Systems with Statecharts*, McGraw-Hill.

Harrison, A. (1995) Business Processes: Their Nature and Properties. In: Burke, G. and Peppard, J., (Eds.) *Examining Business Process Re-engineering: Current Perspectives and Research Directions*, Kogan Page.

Harrison, N., Foote, B. and Rohnert, H. (2000) *Pattern Languages of Program Design 4*, Addison-Wesley.

Harrison, W. and Ossher, H. (1993) *Subject-Oriented Programming (a critique of pure objects)*, Proceedings of OOPSLA 93.

Hay, D.C. (1996) *Data Model Patterns: Conventions of Thought*, Dorset House.

Henderson, P. (1998) Laws for Dynamic Systems. In: Proceedings of International Conference on Software Maintenance, IEEE.

Henderson, P. (2000a) Business Processes, Legacy Systems, and a Fully Flexible Future. In: Henderson, P., (Ed.) *Systems Engineering For Business Process Change*, Springer-Verlag.

Henderson, P. (2000b) *Systems Engineering For Business Process Change*, Springer-Verlag.

Henderson, P. (2001) *Systems Engineering For Business Process Change, volume 2*, Springer-Verlag.

Henning, M. and Vinoski, S. (1999) *Advanced CORBA Programming with C++*, Addison-Wesley.

Highsmith, J.A. (1997) *Messy, Exciting, and Anxiety-Ridden: Adaptive Software Development*, American Programmer, 10, 4.

Highsmith, J.A. (1998) *Order For Free*, Software Development, 6, 3.

Highsmith, J.A. (1999a) *Adaptive Management: Patterns for the E-Business era*, Cutter IT Journal, September.

Highsmith, J.A. (1999b) *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House Publishing.

Highsmith, J.A. (1999c) *Beyond Optimizing*, Software Development, 10, 9.

Hong, Z., Xu, J. and Bennett, K. (2001) An Abstract Architecture for Dependable and Distributed Applications. In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change, volume 2*, Springer-Verlag.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*, Addison-Wesley.

Jacobson, I., Jonsson, P. and Griss, M. (1997) *Software Reuse: Architecture, process and organization for business success*, Addison-Wesley.

Jeffries, R., Anderson, A. and Hendrickson, C. (2001) *Extreme Programming Installed*, Addison-Wesley.

Jones, C. (1991) *Applied Software Measurement*, McGraw-Hill.

JPython (2001) *JPython Documentation*, http://www.jpython.org.

Junit (2000) *Junit Test Framework*, http://www.junit.org.

Kent, S. (1997) *Constraint Diagrams: Visualizing Invariants in Object Oriented Models*, Proceedings of OOPSLA 97, ACM Press.

Kent, S. and Gil, J. (1998) Visualising Action Contracts in OO Modelling. *IEE Proceedings Software* 145, 2-3

Kiczales, G., des Rivieres, J. and Bobrow, D.G. (1991) *The Art of the Metaobject Protocol*, MIT Press.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J. (1997) *Aspect-Oriented Programming*, Proceedings of ECOOP97, Springer-Verlag.

Kotter, J.P. (1996) *Leading Change*, Harvard Business School Press.

Lauder, A. (1999a) *EventPorts*, PhDOOS Workshop, ECOOP 99, Lisbon.

Lauder, A. (1999b) Migrating to Components. In: Griss, M., (Ed.) *OOPLSA 99 Workshop on Making the Transition to Component-Based Enterprise Software Development*,

Lauder, A. (1999c) Pluggable Factory in Practice. *C++ Report* (October): SIGS.

Lauder, A. (1999d) *Responding to Petrification*, 7th SEBPC Legacy Workshop, ICSM 99.

Lauder, A. and Kent, S. (1998) Precise Visual Specification of Design Patterns. In: Jul, E., (Ed.) *Proceedings of ECOOP '98*, Springer-Verlag.

Lauder, A. and Kent, S. (1999a) EventPorts: Preventing Legacy Componentware. In: Atkinson, C., (Ed.) *Proceedings of 3rd International Enterprise Distributed Object Computing Conference (EDOC 99)*, IEEE Press.

Lauder, A. and Kent, S. (1999b) Two-Level Modeling. In: Chen, J., Lu J. and Meyer, B., (Eds.) *Proceedings of TOOLS Asia 99*, IEEE Computer Society.

Lauder, A. and Kent, S. (2000a) *A Pattern is a Metaphor in a World of Straight Lines*, unpublished manuscript, email: anthony@lauder.u-net.com.

Lauder, A. and Kent, S. (2000b) Legacy System Anti-Patterns and a Pattern-Oriented Migration Response. In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change*, Springer-Verlag.

Lauder, A. and Kent, S. (2000c) *Modeling Reusable Product-Line Designs*, unpublished manuscript, email: anthony@lauder.u-net.com.

Lauder, A. and Kent, S. (2000d) *Statecharts for Business Process Modelling*, Proceedings of International Conference for Enterprise Information Systems (ICEIS 2000).

Lauder, A. and Kent, S. (2001a) More Legacy System Patterns. In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change, volume 2*, Springer-Verlag.

Lauder, A. and Kent, S. (2001b) Statecharts for Business Process Modelling. In: Sharp, B., Filipe, J. and Cordeiro, J., (Eds.) *Enterprise Information Systems*, Kluwer Academic Publishers.

Lauder, A. and Lind, M. (1999) *Legacy Systems: Assets or Liabilities?*, Technical Report, University of Boras, Sweden.

Lawson, B. (1997) *How Designers Think: The Design Process Demystified (3rd edition)*, Architectural Press.

Lehman, M.M. (1980) *Programs, Life-Cycles, and the Laws of Program Evolution*, Proceedings of IEEE.

Lehman, M.M. (1997) *Laws of Software Evolution Revisited*, Proceedings of EWSPT 96, Springer-Verlag.

Levy, S. (1994) *Hackers*, Penguin Books.

Lientz, B.P. and Swanson, E.B. (1980) *Software Maintenance Management*, Addison-Wesley.

Lopes, C., Bergmans, L., Black, A. and Kiczales, G. (1999) *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP 99*, Springer-Verlag.

Lopes, C., Mens.K., Tekinerdogan, B. and Kiczales, G. (1997) *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP 97*, Springer-Verlag.

Lopes, C., Tekinerdogan, B., de Meuter, W. and Kiczales, G. (1998) *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP 98*, Springer-Verlag.

Macaulay, S. (1992) Non-Contractual Relations in Business: A Preliminary Study. In: Granovetter, M. and Swedberg, R., (Eds.) Westview Press.

Marsden, P.V. (1987) Elements of Interactor Dependence . In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Martin, R., Riehle, D. and Buschmann, F. (1998) *Pattern Languages of Program Design 3*, Addison-Wesley.

Masiero, P.C. and Braga, R.T.V. (1999) *Legacy Systems Reengineering Using Software Patterns*, Proceedings of the 19th International Conference of the Chilean Computer Science Society, IEEE.

Medina-Mora, R., Winograd, T., Flores, R. and Flores, F. (1992) The Action-Workflow Approach to Workflow Management Technology. In: *Proceedings of 4th Conference on CSCW*, ACM Press.

Microsoft (2001) *.NET*, http://msdn.microsoft.com/NET.

Millea, T., Edwards, J. and Coutts, I. (2001) An Evolutionary Systems Model. In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change, volume 2*, Springer-Verlag.

Miller, H.W. (1997) *Reengineering Legacy Software Systems*, Digital Press.

Mitleton-Kelly, E. (2000a) Complexity: Partial Support for BPR? In: Henderson, P., (Ed.) *Systems Engineering For Business Process Change*, Springer-Verlag.

Mitleton-Kelly, E. and Papaefthimiou, M. (2000b) Co-evolution and an Enabling Infrastructure: A Solution to Legacy? In: Henderson, P., (Ed.) *Systems Engineering For Business Process Change*, Springer-Verlag.

Mitleton-Kelly, E. and Papaefthimiou, M.-C. (2001) Co-Evolution of Diverse Elements Interacting Within a Social Ecosystem. In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change, volume 2*, Springer-Verlag.

Molm, L.M. (1987) Linking Power Structure and Power Use . In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Morgan, G. (1996) Unfolding Logics of Change: Organizations as Flex and Transformation. In: *Images of Organizations*, Sage.

Mumford, E. (1995) Creative Change or Constructive Chaos: Business Process Re-engineering vs. Socio-Technical Design. In: Burke, G. and Peppard, J., (Eds.) *Examining Business Process Re-engineering: Current Perspectives and Research Directions*, Kogan Page.

Nielson, J. (1999) *Designing Web Usability*, New Riders.

O Cinneide, M. (2000) *Automated Application of Design Patterns: A Refactoring Approach (PhD Thesis)*, Trinity College, University of Dublin.

Opdyke, W.F. (1992) *Refactoring Object Oriented Frameworks (PhD thesis)*, University of Illinois at Urbana-Champaign.

Osborne, W.M. and Chikofsky, E.J. (1990) Fitting Pieces to the Maintenance Puzzle. *Software* (November): IEEE.

Ould, M.A. (1995) *Business Processes: Modelling and Analysis for Re-engineering and Improvement*, Wiley.

Parnas, D.L. (1976) On the design and development of program families . *IEEE Transactions on Software Engineering* March

Peterson, A.S. and Stanley, Jr.J.L. (1994) Mapping a Domain Model and an Architecture to a Generic Design (CMU/SEI-94-TR-8). Software Engineering Institute, Carnegie Mellon University.

Petrie, H.G. (1979) Metaphor and Learning. In: Ortony, A., (Ed.) *Metaphor and Thought*, Cambridge University Press.

Pigoski, T.M. (1997) *Practical Software Maintenance*, Wiley.

Pree, W. (1994) *Design Patterns for Object-Oriented Software Development*, ACM Press.

Prieto-Dias, R. and Arango, G. (1991) *Domain Analysis and Software Systems Modelling*, IEEE Computer Society Press.

Ramage, M. and Bennett, K. (1998) *Maintaining Maintainability*, Proceedings of International Conference on Software Maintenance, IEEE.

Ramage, M., Brooke, C., Bennett, K. and Munro, M. (2000a) Combining Organisational and Technical Change in Finding Solutions to Legacy Systems. In: Henderson, P., (Ed.) *Systems Engineering For Business Process Change*, Springer-Verlag.

Ramage, M. and Munro, M. (2000b) Its Not just about Old Software: A Wider View of Legacy Systems. In: Henderson, P., (Ed.) *Systems Engineering For Business Process Change*, Springer-Verlag.

Randall, D., Rodden, T., Rouncefield, M. and Sommerville, I. (2001) Remeberance of Designs Past: Legacy Data, Organisational Memory and Distributed Design. In: Henderson, P., (Ed.) *Systems Engineering for Business Process Change, volume 2*, Springer-Verlag.

Rank, S., Bennett, K. and Glover, S. (2000) FLEXX: Designing Software for Change Through Evolvable Architectures. In: Henderson, P., (Ed.) *Systems Engineering For Business Process Change*, Springer-Verlag.

Reengineering Center (1995) Perspectives on Legacy System Reengineering (Draft 0.3). Software Engineering Institute, Carnegie Mellon University.

Reifer, D.J. (1997) *Practical Software Reuse: Strategies for Introducing Reuse Concepts into Your Organization*, Wiley.

Riehle, D. (1996) The Event Notification Pattern - Integrating Implicit Invocation with Object-Orientation. *Theory and Practice of Object Systems* 2, 1

Riehle, D., Siberski, W., Baumer, D., Megert, D. and Zillinghoven, H. (1998) Serializer. In: Martin, R., Riehle, D. and Buschmann, F., (Eds.) *Pattern Languages of Program Design 3*, Addison-Wesley.

Rising, L. (2000) Customer Interaction Patterns. In: Harrison, N., Foote, B. and Rohnert, H., (Eds.) *Pattern Languages of Program Design 4*, Addison-Wesley.

Roberts, D.B. (1999) *Practical Analysis for Refactoring (PhD thesis)*, University of Illinois at Urbana-Champaign.

Rumbaugh, J., Blaha, M., Premberlani, W., Eddy, F. and Lorensen, W. (1991) *Object-Oriented Modeling and Design*, Prentice Hall.

Rumbaugh, J., Jacobson, I. and Booch, G. (1999) *The Unified Modeling Language Reference Manual*, Addison-Wesley.

Schmidt, D. (1995) Reactor: An Object Behavioral Pattern for Concurrent Event Multiplexing and Event Handler Dispatching. In: Coplien, J.O. and Schmidt, D., (Eds.) *Pattern Languages of Program Design* , Addison-Wesley.

Schmidt, D. (1999) *The ADAPTIVE Communication Environment*, http://www.cs.wustl.edu/~schmidt/ACE.html.

Schwaber, K. (1995) *The Scrum Development Process*, OOPSLA 95 Workshop on Business Object Design and Implementation.

Schwaber, K. (1996) *Controlled Chaos: Living on the Edge*, American Programmer (spring issue).

ScrumOnline (2000a) *Scrum Case Studies (1-8)*, http://www.scrumonline.com.

ScrumOnline (2000b) *Scrum Reference*, http://www.scrumonline.com.

Searle, J.R. (1969) *Speech Acts. An essay in the philosophy of language*, Cambridge University Press.

Seiter, L.M. (1996) *Design Patterns for Managing Evolution (PhD thesis)*, Northeastern University.

Sharon, D. (1996) *Meeting the Challenges of Software Maintenance*, IEEE Computer Society Press.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Shiff, R. (1979) Art and Life: A Metaphoric Relationship. In: Sacks, S., (Ed.) *On Metaphor*, University of Chicago Press.

Signore, R., Creamer, J. and Stegman, M.O. (1995) *The ODBC Solution: Open Database Connectivity in Distributed Environments*, McGraw-Hill.

Sneed, H.M. (1995) Planning the Reengineering of Legacy Systems. *Software* (January): IEEE.

SoftWired AG (1999) *iBus*, http://www.softwired-inc.com.

Stevens, P. and Pooley, R. (1998) *Systems Reengineering Patterns*, Proceedings of Foundations of Software Engineering, ACM.

Sticht, T.G. (1979) Educational Uses of Metaphor. In: Ortony, A., (Ed.) *Metaphor and Thought*, Cambridge University Press.

Suchman, L. (1987) *Plans and Situated Actions*, Cambridge University Press.

Swanson, E.B. (1976) *The Dimensions of Maintenance*, Proceedings of International Conference on Software Maintenance, IEEE.

Swanson, E.B. and Beath, C.M. (1989) *Maintaining Information Systems in Organizations*, Wiley.

Szyperski, C. (1998) *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley.

Talarian (1999) *SmartSockets*, http://www.talarian.com.

Taylor, P. (1999) *Hackers*, Routledge.

Taylor, P. (2000) Capable, Productive, and Satisfied: Some Organizational Patterns for Protecting Productive People. In: Harrison, N., Foote, B. and Rohnert, H., (Eds.) *Pattern Languages of Program Design 4*, Addison-Wesley.

Thorp, J. (1998) *The Information Paradox* , McGraw Hill.

Tibco (1999) *Rendezvous Information Bus*, http://www.tibco.com.

Uschold, M. and Gruninger, M. (1996) Ontologies: Principles, Methods, and Applications. *The Knowledge Engineering Review* 11, 2

van Deursen, A., Klint, P. and Verhoe, C. (1999) Research Issues in the Renovation of Legacy Systems. In: Finance, J.-P., (Ed.) *Fundamental Approaches to Software Engineering (FASE 99)*, Springer-Verlag.

van Deursen, A., Kuipers, T. and Moonen, L.M.F. (2001) Legacy to the Extreme. In: Succi, G. and Marchesi, M., (Eds.) *Extreme Programming Examined*, Addison-Wesley.

van Reijswoud, V.E. and van der Rijst, N.B.J. (1999) Modelling Business Communication as a Foundation for Business Process Redesign. In: *Proceeding of the 28th International Conference on Systems Science*, IEEE Computer Society Press.

Veale, T. (1995) *Symbolic and Connectionist Issues in Metaphor Interpretation (PhD thesis)*, Dublin City University.

Veale, T. (1998) *A Survey of the Metaphor Field*, http://www.compapp.dcu.ie/~tonyv/survey_frame.html.

Vlissides, J. (1997a) Multicast. *C++ Report* (September): SIGS.

Vlissides, J. (1997b) Patterns: The Top Ten Misconceptions. *Object Magazine* (March): SIGS.

Vlissides, J. (1998) Pluggable Factory, Part 1. *C++ Report* (November): SIGS.

Vlissides, J. (1999) Pluggable Factory, Part 2. *C++ Report* (January): SIGS.

Vlissides, J., Coplien, J.O. and Kerth, N. (1996) *Pattern Languages of Program Design 2*, Addison-Wesley.

Vollman, T.E. and Garbajosa-Sopena, J. (1996) *CASE Tool Support for Software*, Proceedings of the European Space Agency Software Product Assurance Symposium.

Warmer, J. and Kleppe, A. (1998) *The Object Constraint Language: precise modeling with UML*, Addison-Wesley.

Warren, I. (1999) *The Renaissance of Legacy Systems*, Springer.

Weiderman, N.H., Bergey, J.K., Smith, D.B. and Tilley, S.R. (1997) Approaches to Legacy System Evolution. CMU/SEI-97-TR-014, Software Engineering Institute, Carnegie Mellon University.

Weinberg, G.M. (1994) *Quality Software Management: Congruent Action*, Dorset House Publishing.

Weiss, D.M. and Lai, C.T.R. (1999) *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley.

Wiki Community (2000) *CthreeProjectTerminated*, http://c2.com/wiki?CthreeProjectTerminated.

Williams, L.A. (2000) *The Collaborative Software Process (PhD thesis)*, University of Utah.

Wills, A. (1998) *Formal Methods and Patterns*, http://c2.com/wiki?FormalMethodsAndPatterns.

Winograd, T. and Flores, F. (1987) *Understanding Computers and Cognition: A new foundation for design*, Addison-Wesley.

Wittgenstein, L. (1958) *Philosophical Investigations*, Basil Blackwell.

Woolf, B. (1998) Null Object. In: Martin, R., Riehle, D. and Buschmann, F., (Eds.) *Pattern Languages of Program Design 3*, Addison-Wesley.

Woolf, B. (2000) Abstract Class. In: Harrison, N., Foote, B. and Rohnert, H., (Eds.) *Pattern Languages of Program Design 4*, Addison-Wesley.

Yacoub, S.M. and Ammar, H.H. (2000) Finite State Machine Patterns. In: Harrison, N., Foote, B. and Rohnert, H., (Eds.) *Pattern Languages of Program Design 4*, Addison-Wesley.

Yamagishi, T. (1987) An Exchange Theoretical Approach to Network Positions. In: Cook, K., (Ed.) *Social Exchange Theory*, SAGE Publications.

Yourdon, E. and Constantine, L. (1978) *Structured Design*, Yourdon Press.