1-1-2014

An Empirical Comparison of Reuse in Embedded and Nonembedded Systems

Julia F. Varnell-Sarjeant
University of Denver

An Empirical Comparison of Reuse in Embedded and Nonembedded Systems

A Dissertation

Presented to the Faculty

of The Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

in Partial Fulfillment

of the Requirements for the Degree

of Doctor of Philosophy

by

Julia F. Varnell-Sarjeant

June, 2014

Advisor: Dr. Anneliese Andrews

Author: Julia F. Varnell-Sarjeant
Title: An Empirical Comparison of Reuse in Embedded and Nonembedded Systems
Advisor: Dr. Anneliese Andrews
Degree Date: June, 2014

Abstract

High-quality software, delivered on time and budget, constitutes a critical part of most products and services in modern society. Our government has invested billions of dollars to develop software assets, often to redevelop the same capability many times. Recognizing the waste involved in redeveloping these assets, in 1992 the Department of Defense issued the Software Reuse Initiative.

The vision of the Software Reuse Initiative was "To drive the DoD software community from its current "re-invent the software" cycle to a process-driven, domain-specific, architecture-centric, library-based way of constructing software." Twenty years after issuing this initiative, there is evidence of this vision beginning to be realized in nonembedded systems. However, virtually every large embedded system undertaken has incurred large cost and schedule overruns. Investigations into the root cause of these overruns implicates reuse. Why are we seeing improvements in the outcomes of these large scale nonembedded systems and worse outcomes in embedded systems? This question is the foundation for this research.

The experiences of the Aerospace industry have led to a number of questions about reuse and how the industry is employing reuse in embedded systems. For example, does reuse in embedded systems yield the same outcomes as in nonembedded systems? Are the outcomes positive? If the outcomes are different, it may indicate that embedded systems should not use data from nonembedded systems for estimation. Are embedded systems using the same development approaches as nonembedded systems? Does the development approach make a difference? If embedded systems develop software differently from nonembedded systems, it may mean that the same processes do not apply to both types of systems.

What about the reuse of different artifacts? Perhaps there are certain artifacts that, when reused, contribute more or are more difficult to use in embedded systems. Finally, what are the success factors and obstacles to reuse? Are they the same in embedded systems as in nonembedded systems?

The research in this dissertation is comprised of a series of empirical studies using professionals in the aerospace and defense industry as its subjects. The main focus has been to investigate the reuse practices of embedded systems professionals and nonembedded systems professionals and compare the methods and artifacts used against the outcomes. The research has followed a combined qualitative and quantitative design approach. The qualitative data were collected by surveying software and systems engineers, interviewing senior developers, and reading numerous documents and other studies. Quantitative data were derived from converting survey and interview respondents' answers into coding that could be counted and measured.

From the search of existing empirical literature, we learned that reuse in embedded systems are in fact significantly different from nonembedded systems, particularly in effort in model based development approach and quality where the development approach was not specified.

The questionnaire showed differences in the development approach used in embedded projects from nonembedded projects, in particular, embedded systems were significantly more likely to use a heritage/legacy development approach. There was also a difference in the artifacts used, with embedded systems more likely to reuse hardware, test products, and test clusters. Nearly all the projects reported using code, but the questionnaire showed that the reuse of code brought mixed results. One of the differences expressed by the respondents to the questionnaire was the difficulty in reuse of code for embedded systems when the platform changed.

The semistructured interviews were performed to tell us why the phenomena in the review of literature and the questionnaire were observed. We asked respected industry professionals, such as senior fellows, fellows and distinguished members of technical staff,

about their experiences with reuse. We learned that many embedded systems used heritage/legacy development approaches because their systems had been around for many years, before models and modeling tools became available. We learned that reuse of code is beneficial primarily when the code does not require modification, but, especially in embedded systems, once it has to be changed, reuse of code yields few benefits. Finally, while platform independence is a goal for many in nonembedded systems, it is certainly not a goal for the embedded systems professionals and in many cases it is a detriment. However, both embedded and nonembedded systems professionals endorsed the idea of platform standardization.

Finally, we conclude that while reuse in embedded systems and nonembedded systems is different today, they are converging. As heritage embedded systems are phased out, models become more robust and platforms are standardized, reuse in embedded systems will become more like nonembedded systems.

Acknowledgements

This research is dedicated to the women who made it possible. To Dr. Andrews for her patient and tireless guidance. To Dorothy McKinney and Willa Pickering for their inspiration, endless support and wise advice. These three women showed me that you are never too old or too female to accomplish your dreams. To Dr. Andreas Stefik and Joe Lucente for their contributions to the survey analysis. To my husband, who inspired and supported me. And to my children, who have dreams of their own.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reuse has been advocated as facilitating faster and cheaper development with higher levels of quality. It is claimed to reduce effort, cost and to increase quality [4], improve maintainability, reduce risk, shorten life cycle time, lower training costs, and achieve better software interoperability [101]. "If a software package has been executing error-free in the field for an extended period, under widely varying, perhaps stressful, operating conditions, and it is then applied to a new situation, one strongly expects that it should work error free in this new situation [4]." IEEE [65] states that the major benefits that systematic reuse can deliver are:

- Increase software productivity
- Shorten software development and maintenance time
- Reduce duplication of effort
- Move personnel, tools, and methods more easily among projects
- Reduce software development and maintenance costs
- Produce higher quality software products
- Increase software and system dependability
- Improve software interoperability and reliability

Some interpret the IEEE Standard to be saying that benefits realized in nonembedded systems are extensible to embedded systems. However, there may be some evidence that several of these claims are not true. One particularly nagging question is whether embedded systems benefit from reuse in the same ways that nonembedded systems do. Some companies are establishing reuse processes and practices that are imposed on all systems [65]. IEEE has claimed in its reuse standard that "One major problem encountered by organizations attempting to practice reuse is that reuse is simply missing from their life cycle processes. To harness the benefits of reuse, an organization must incorporate reuse throughout its system and software processes. An organization that creates systems and software first and considers reuse second may not fully benefit from reuse practices [65]."

The US Aerospace Industry was an early advocate of reuse. In July of 1992, the DoD released the DoD Reuse Initiative: Vision and Strategy [14]. This is why we started the investigation with the studies from 1992 and later. The US government has invested heavily in reuse, e.g. the Control Channel Toolkit (CCT) in 1997 and Global Broadcasting Service (GBS) beginning in 1998. Hence, one would expect large scale planned reuse. Many US government requests for proposal contain a reuse requirement including quantifying expected savings from reuse.

Yet there are those who question whether successful reuse strategies work equally well for all types of systems. Those supporting reuse cited the existence of working, already developed assets that were performing similar capabilities. Those supporting new development made the claim that when you ask a product to do what it was not designed to do, the costs of modification and maintenance exceed the savings of reuse. Further, some claim that embedded systems are fundamentally different from nonembedded systems, and hence, successful reuse needs different strategies. Reasons include:

- Embedded systems are written and optimized directly to the target processors. If the reused products are to be run on a different platform, much of the code will have to be modified to accommodate the standard of the different platform.

- Throughput, processing, and timing of embedded systems are critical to system performance, and thus to mission success.
- Much software available for reuse is old, sometimes obsolete. It may have been written to processors that are no longer supported by the vendors. In order to make that software work, it has to be reoptimized to newer processors.
- Platforms running the software are changing faster than the software.
- Embedded systems are not allowed to deploy with code that is not needed (many reuse assets include more general solutions).

Others think that software is software, and the same reuse strategies can be used on either type of system. It is, hence, unclear whether reuse is different for embedded versus nonembedded systems.

The aerospace industry develops software for both embedded and nonembedded systems. However, in Aerospace, not all reuse experiences have been as successful as expected. Some major projects experienced large overruns in time, budget, as well as inferior performance, at least in part, due to the wide gap between reuse expectations and reuse outcomes [115, 132]. This seemed to be especially the case for embedded systems. In many US government customer shops, reuse became a red flag when awarding contracts. Bidders found that they could bid against reuse as successfully as for it.

Some engineers believed that one of the root causes of the disconnect between reuse expectations and reuse realization was that too often estimated savings from reuse came from projects that were not the same as the project being bid (c.f. Chapter 5). In particular, the reuse estimates for nonembedded systems were applied to embedded systems. There was an ongoing debate as to whether embedded systems and nonembedded systems were, in fact, analogous. Many systems and software engineers, especially those who worked on embedded systems, claimed that reuse could not be successfully used in their systems because the code was optimized to particular processors which may not be used in the new project. Many of the embedded systems software engineers claimed that reusing software

was more costly than building the system from scratch. Further, many claimed that, in particular, trying to use model based development (derived from the Initiative) was more costly than using other development approaches.

We undertook this investigation in an environment in which reuse was mandated in bidding and implementation. However, bidding savings from reuse was difficult, in that we could not locate metrics from similar embedded programs as justification for the claims of expected savings. Given the likelihood that there might be a difference based on types of systems, using savings metrics from nonembedded systems seemed unreliable. Once the project was started, the absence of these metrics and processes in similar systems led to importing processes from nonsimilar projects. The outcomes were often less than expected, sometimes the outcomes were believed to be worse than if reuse had not been employed. So the researcher wondered, are embedded systems truly different from nonembedded systems when it comes to reuse? Specifically, does reuse effectiveness vary between embedded and nonembedded software? Do embedded systems projects employ the same development and reuse strategies? Are they reusing the same types of software and hardware artifacts? If yes, can reuse be implemented in a way that is beneficial to embedded systems? If not, why are the outcomes so often disappointing?

To shed light on these questions, a survey was designed to collect information on reuse success and challenges for embedded vs nonembedded systems, and to compare reuse outcomes for different development strategies and their related reuse artifacts.

This chapter describes the background (Section 1.1). the problem statement (Section 1.2) and the context for the research (Section 1.2). In addition, this chapter also presents research questions (Section 1.2.2), the contributions (Section 1.2.3) and finally the thesis oranization (Section 1.3).

1.1 Problem Statement

High-quality software, delivered on time and budget, constitutes a critical part of products and services in modern society. The Department of Defense has recognized that developing and maintaining high quality systems is an expensive endeavor and has looked for ways to curtail these costs. Looking to leverage existing assets, the DoD Software Reuse Initiative developed a set of goals, priorities and processes to reuse existing assets. This has worked well in a number of large nonembedded systems. However, embedded systems have not fared as well. Almost every large DoD embedded system undertaken over the past decade has experienced serious cost and schedule overruns. A major root cause of these overruns has been identified as reuse practices [115]. This is a major concern as existing assets are aging and need to be replaced. The difference in outcomes between embedded and nonembedded systems suggests that perhaps embedded and nonembedded systems are different, requiring different treatment. The questions are, is reuse only valuable in nonembedded systems? Can we not reuse existing assets in embedded systems? Or do we simply need to do embedded system reuse differently? If the answer to the last question is yes, then what do we need to do differently to have success in reuse in embedded systems? Do they need to use different development approaches? Do they need to reuse different assets? Do they need different processes?

This thesis will investigate reuse in embedded and nonembedded systems. It will see what research has already been done to compare the two types of systems looking for similarities and differences. It will compare development methods and artifacts, and attempt to identify the impact on the outcomes related to each.

1.1.1 Research Context

The research in this thesis relies on quantitative and qualitative empirical studies of embedded and nonembedded systems, as well as surveys and interviews with professionals in the defense and Aerospace industries.

The DoD and, by extension, the United States government, issued its reuse initiative in 1992. The researcher has observed that since that time, requests for proposals have required a reuse program be included in any contractor offerings. Sufficient time has passed to be able to judge the effectiveness of the initiative and identify weaknesses.

One noticeable weakness has been the disappointing performance of embedded systems. Among the embedded systems that employed disappointing results were, SBIRS, GPS II F, Presidential Helicopter, Global Broadcasting System, GOES N and Littoral Combat System. There were many more. These projects were from all of the major defense contractors, so the common element is not the company. They were a spectrum of types of products (satellite, helicopter, underwater), as well as from different branches of government, so the cause is not either in product or branch. In fact, the only element in common is the embedded nature of the systems.

It is within this backdrop that we ask whether there is a difference in reuse between embedded and nonembedded systems.

1.1.2 Research Questions

The overall research goal (RG) for all studies carried out as part of this thesis was to investigate the advantages/disadvantages of software reuse and the reasons behind it, by analyzing reuse in embedded and nonembedded systems. Then, based on these insights, propose specific reuse guidelines (as an example of improvements) software practitioners regarding the approaches and artifacts that bring about the best outcomes in each type of system.

We performed empirical studies (i.e., survey and semi-structured interviews) in large aerospace companies. The objective for this research was to investigate the commonalities and differences between embedded and nonembedded systems software reuse. We propose that the differences we found should be considered in the development of company processes and mandates. Studying software reuse increases our understanding of the relationship between reuse in embedded systems and reuse in nonembedded systems. It helps

us determine early in a project the best approaches to take and the products to reuse. This benefits the research community, practitioners and customers. The former gains deeper insight into benefits and challenges of software reuse with respect to the types of systems being developed. The latter gain insight into how to implement software reuse in a way to obtain better outcomes and encounter less frustration for both developers and managers. Customers get better estimates of cost at the beginning to enable better budgeting, and products developed on time and on schedule.

The thesis presents three studies, and we have formulated four research questions that explore our overall goal. In order to go from our overall goal to specific studies and research questions, we have formulated the following questions:

- RQ1: How do outcomes for reuse in embedded systems compare with the outcomes for nonembedded systems?
- RQ2: How do development approaches for embedded systems compare with development approaches for nonembedded systems?
- RQ3: Do embedded systems reuse the same artifacts as nonembedded systems, and how do the outcomes for reuse of artifacts compare?
- RQ4: Are the success factors and obstacles the same for embedded systems the same as for nonembedded systems?

The main types of research design included are document analysis; survey; and interviews. In each case, qualitative data was collected and analyzed, then converted to quantitative data for analysis.

Figure 1.1 shows the goals, questions and metrics used to guide the research.

We wanted to identify what reuse strategies work best (most successful) in large embedded systems and software and how they compare between nonembedded systems and

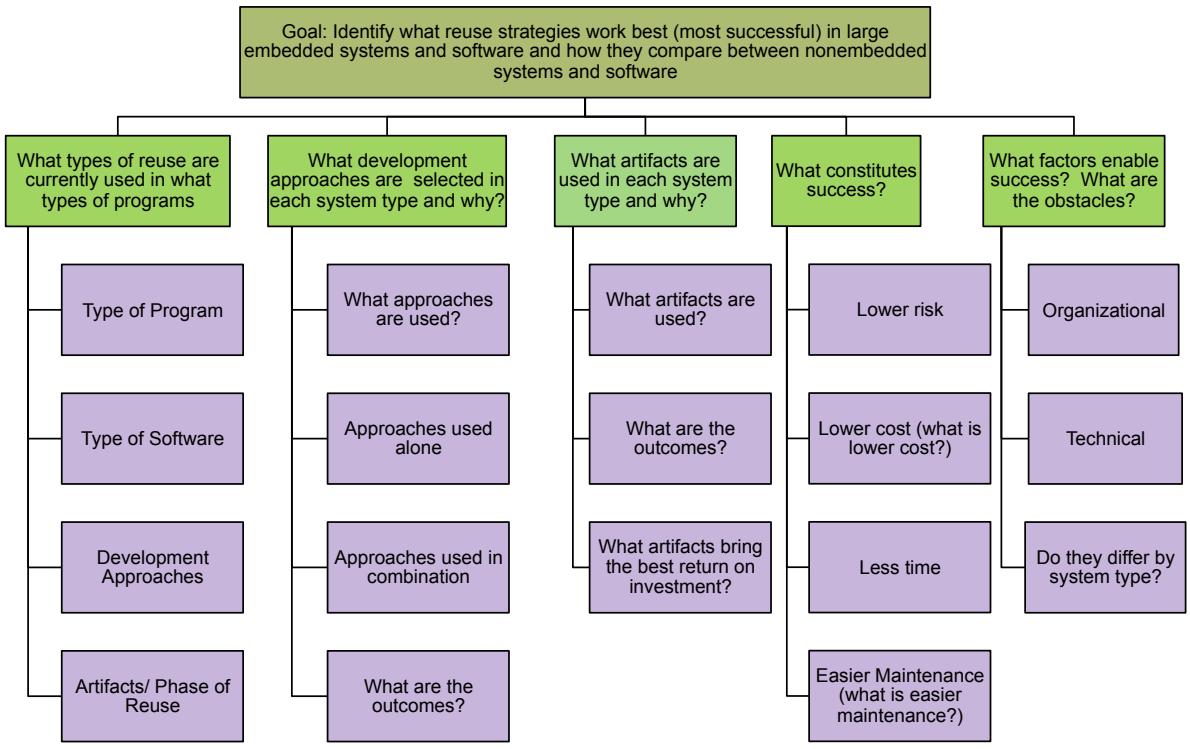


Figure 1.1: Goal Question Metric Diagram

software. To do this, we begin with an examination of existing literature to see if the discipline at large has investigated whether embedded systems are the same or different in their reuse practices and outcomes. The examination of literature is followed by a questionnaire for software and systems practitioners in the Aerospace industry to identify their current practices. The questionnaire is followed by semistructured interviews to see if we can identify success factors and obstacles to successful reuse. If, in fact, it turns out that embedded systems are different from nonembedded systems, we will offer recommendations of how embedded systems need to employ reuse differently.

1.1.3 Contributions and Papers

The main contributions of this research are:

- C1: Identification of differences/similarities between embedded and nonembedded systems development approaches and their outcomes.

- C2: Identification of differences/similarities of reused artifacts by embedded and nonembedded systems and the return on investment for each type of artifact .
- C3: Identification of differences/similarities between embedded and nonembedded systems of the success factors and obstacles and ways to employ the success factors better and overcome the obstacles
- C4: Identification of possible software reuse improvements.

Papers developed for submission to peer reviewed publications include:

- Reflections on the History of Software Reuse for Embedded Systems in the Defense and Aerospace Industry
- Comparing Reuse in Different Development Approaches For Embedded vs Non-Embedded Systems
- Comparing Development Approaches and Reuse Strategies: An Empirical Evaluation of Developer Views
- Reuse Practices in Aerospace: Structured Interviews

1.2 Organization

Figure 1.2 shows the structure of this document.

Chapter one discusses the rationale for the research, the nature of the problem, the context in which the research is performed, and contributions to the current body of knowledge. Chapter two discusses the background of reuse in the aerospace industry Chapter three discusses a review of existing literature. Chapter four covers information obtained through a survey in a major aerospace corporation. Chapter five discusses results from semistructured interviews that reveal information about reasons for reuse success and failure and why

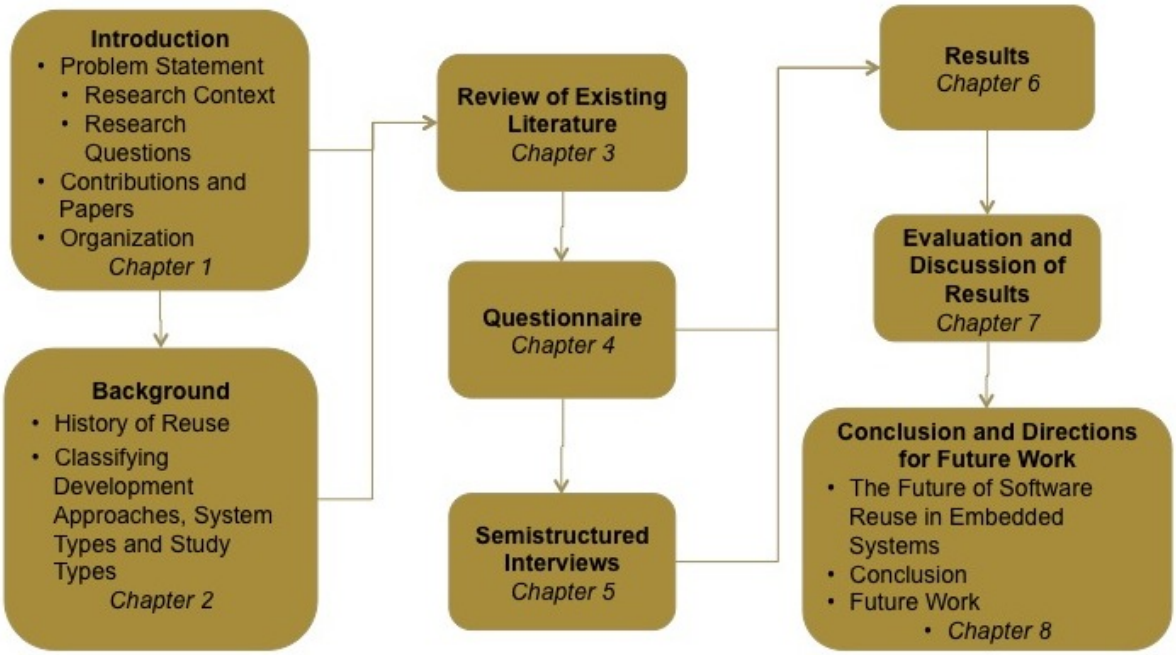


Figure 1.2: Structure of the Dissertation

,

the outcomes in embedded systems have differed from outcomes in nonembedded systems. Chapter six discusses the results of this research and evaluates the results. It also presents the conclusions, the future of software reuse in embedded systems and recommendations for future work.

Chapter 2

Background

2.1 History of Reuse

Reuse has a long history in the defense and aerospace industry, spanning several generations of technology. Throughout this history, software engineers have tried different ways to reuse their software to increase savings and efficiency. Sometimes these reuse methods have kept up with the evolution of software development techniques, sometimes they have not. One of the debates in the software community has been whether software development strategies and corresponding reuse strategies are the same for embedded systems and nonembedded systems. This chapter reflects on some of the success factors and barriers to successful reuse in embedded systems. It covers early reuse, government mandates and modern reuse technology like Component Based and Product Line development approaches. It looks at different artifacts that are reused, and the future of reuse of software and system products, and how that evolution may differ between embedded and nonembedded systems.

2.2 Introduction

The aerospace industry was an early advocate of reuse. As reuse evolved, what was being reused, how it was reused and why it could be reused should have evolved as well, especially in light of government reuse mandates. The government invested heavily in

reuse, e.g. the Control Channel Toolkit (CCT) in 1997 and Global Broadcasting Service (GBS) beginning in 1998. Hence, one would expect large scale planned reuse. This section explores the history of reuse in aerospace and attempts to shed light on reuse evolution via published reports, interviews, internal documents and summary results from an internal survey.

For decades, managers and software developers have looked at the volume of existing code and been convinced that we could save a lot of money if we could reuse it [40]. The industry has studied and experimented with ways to reuse software products, including: requirements, architecture, design, and test products. However, the expected savings in productivity and reliability simply haven't been realized ([4], [111], [47], [7]).

Intuitively, reusing existing assets should save money. After all, they don't have to be created. However, they often have to be modified, so the question becomes: is it faster to create from scratch or to re-create from existing products? In many situations, the software products cannot be reused intact. "Even when the components are in hand, significant problems often remain, because the chosen parts do not fit well together [47]."

> "The cultural shift required by management to understand the issues and culture of the software engineers was our greatest challenge. Management expectations were originally high that software engineers could possess the same domain knowledge as systems engineers, and this was simply not the case. The mindset of someone with a master's degree in mechanical or electrical engineering, especially if that degree was granted more than 10 years ago, is fundamentally different from the mindset of a contemporary software engineer [24]."

2.3 Early Development Approaches and Reuse

The development approaches and reuse discussed here reflect the authors' personal experiences.

2.3.1 Ad Hoc Reuse

The first software development approach using reuse in embedded systems was ad hoc reuse, i.e. the reuse of tidbits of code that happened to be available. Reuse in ad hoc development is "Unplanned and opportunistic reuse that fails to meet the full potential of reuse. It is performed with little or no planning or commitments to produce, broker or consume assets [88]." Every software developer comes out of school with a set of routines he has already implemented. "Individuals and small groups have always practiced ad hoc software reuse [40]." This code was not designed to be reused, but it satisfied a requirement for a given capability. This is known as "ad hoc" reuse - software salvaging from existing applications or products. This first development process with software reuse has existed as long as software has existed. A similar type of "reuse" was the code the developer had created before. In the early days of programming, when developers had to punch cards and enter them into the card reader, most of the reuse was contained in the head of the developer. He knew how to implement the solution because he had implemented it before. However, it was easier to simply rewrite the code from memory than to search through a stack of cards to pull out the cards that contained the code he needed. These routines were largely undocumented, or their documentation was in hard copy and not readily available.

2.3.2 Structured Programming

One of the early deterrents to reuse was unstructured code. In his letter to the Editor of the Communications of the ACM in March of 1968, Edsger Dijkstra stated that "the go to statement has such disastrous effects ... that the go to statement should be abolished from all 'higher level' programming languages [31]." He goes on to say, "it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Values of variables can only be known in the context of that process and are hard to trace when the program thread is continually jumping from one context to another." If a function had to be changed, the developer had to trace through a tangle of "spaghetti code" to find

all lines affected by the change. This letter has been cited as the beginning of structured programming. Structured programming is an element of imperative programming, with its major contribution being the reduction in the reliance on the "go-to" statement. Instead, the functions are combined in three ways, sequencing, selection and iteration. With the introduction of structured programming, the developer could contain all references to a behavior within a function. The context of the variables could be maintained and easily traced. Changes could be found and made more easily, and it was easier for another developer to receive the code and understand what was happening. The increased ability for code to be passed from developer to developer enhanced its reusability.

2.3.3 Libraries

Once it became easier for developers to use and reuse each others' code, it became clear that there was a need to collect and make available commonly used routines. These collections were stored on a server that could be accessed by every developer in the environment, and became known as libraries. Reusing routines from these libraries had dual benefits: often performed functions did not have to be recreated for every software program, and use of these functions ensured a level of commonality across a project.

In the 70's and 80's, the aerospace industry produced a number of compartmentalized systems and constellations, each with its own culture and development approach. These were isolated, and did not interact with each other. In many cases, they solved the same problems with different solutions. There was no cross-pollinization of requirements, design or code. When the government wanted to use the information gathered from these constellations together and operate them from a common ground station, the constellations could not easily integrate with each other.

Initially libraries consisted of basic capabilities, such as frequently used mathematical algorithms. Each constellation had its own library. When the Naval Weapons Laboratory computed precise satellite ephemerides for Navy navigation satellites, these became part of the libraries. In the 90's, a major aerospace company concluded that there were vast

volumes of good code hidden away that could be used on many programs across the corporation. It initiated an effort to collect this code into a corporate library, catalog it, and encourage developers on new programs to consider it in their bid and development. Several years later, this library closed down due to inactivity [91].

2.3.4 Legacy or Heritage Reuse

Soon libraries contained a large numger of routines commonly used by a variety of projects. These routines began to be used en masse as foundations for projects, leading to legacy or heritage reuse. Legacy, or Heritage software development is defined by NASA as: "software products (architecture, code, requirements) written specifically for one project and then, without prior planning during its initial development, found to be useful on other projects [96]." Booch et al [13] added: "We refer to a legacy system as one for which there is a large capital investment that cannot economically or safely be abandoned." As companies were awarded contracts that were follow-on efforts to existing projects, they realized that many capabilities existed. Not only was the code useful, but much of it had already been integrated and tested together.

Heritage software began to consist of large clusters of functions, sometimes to the subsystem level, e.g., in satellite command and control, the capabilities required to guide the satellite did not change significantly even though the payload was completely different. Legacy code became the mainstay of many constellations of satellites, ground systems, signal processing, etc. The new capabilities were known as "one-offs" and only required modification to include changes necessary to achieve the new capabilities. Heritage reuse continues to be widely used today.

2.4 New Technologies and Standards Affecting Reuse

Reuse strategies in embedded systems were also affected by new technologies and standards which were not necessarily developed to encourage higher levels of reuse:

2.4.1 Interface Standards.

One of the difficulties with integrating the routines in the libraries and migrating these routines to other projects was the difference in the interfaces, or the way these routines communicated with each other. The International Organization for Standardization and the International Electrotechnical Commission ISO/IEC 11404:1996 [67], updated in 2006 [68], developed a common set of information technology standards and guidelines to be used in all new and upgraded acquisitions across the Department of Defense (DoD) for sending and receiving information (information transfer standards), for understanding the information (information content and format standards) and for processing that information.

Other standards were provided by Comite Consultatif International Telephonique et Telegraphique (CCITT now known as Telecommunication Standardization Sector of the International Telecommunications Union (ITU-T)(e.g. CCITT v.28) and Aeronautical Radio, Incorporated (ARINC)(e.g.ARINC 653). These standards allowed the developers to produce "black box" products that theoretically would work smoothly with other products with little or no adjustment as long as the appropriate standard was observed.

2.4.2 Object-Oriented Languages.

About the time when libraries and interface standards were being developed, some computer professionals were noticing that these routines were themselves components, or objects, similar to components in hardware. The idea of objects had been around for a while. Computer professionals determined that development of software should use objects as the foundation of their programs. This would come to be known as "object oriented development."

Simula 67, in the 1960's, was the first object-oriented language, developed to create simulations [49, p. 47]. It describes behavior and data. These elements became what was known as a "class." Classes had the ability to encapsulate behavior and define a way for that behavior to communicate with other components.

In 1983, the DoD introduced Ada which targeted large embedded and real-time systems. In 1995, Ada95 added object-oriented features. In 1987 programs were required to use Ada, including Military Strategic and Tactical Relay (MILSTAR), Global Positioning System (GPS) and a number of classified programs. Eventually the C language adopted objects, and C++, introduced in 1982, became the predominant object-oriented language. While in large embedded systems so often developed in aerospace, many of the features of C++ were not useable, such as dynamic memory and stereotypes, the object-oriented nature of C++ was adopted. The resulting language was called Embedded C++, or EC++.

Converting functions from structured programming to Object Oriented programming was expected to be pretty simple. While structured programming consisted of a series of functions, Object Oriented programming consisted of a number of classes made up of attributes and methods. The difference between functions and methods is subtle but important. A function is a piece of code that is called by name and can be passed data to operate on (ie. the parameters) and can optionally return data (the return value). All data that is passed to a function is explicitly passed. A method is a piece of code that is called by name that is associated with an object. A method is implicitly passed the object for which it was called. It is able to operate on data contained within the class. It was argued that functions should be easily converted to methods. However, we are finding that because of these subtle differences, often this conversion can take longer than redeveloping the software from scratch.

2.4.3 Unified Modeling Language (UML)

Once object oriented programming became the state of the art, software engineers began to develop software counterparts to the hardware Computer Aided Design (CAD) programs. UML's roots are Object-oriented Analysis and Design (OOAD), The Object Modeling Technique (OMT) and The Object-oriented Software Engineering method (OOSE) [13]. By modeling interfaces and treating components as actors, UML makes model reuse not only possible but useful for its structural and behavioral models. Further, specific performance

and platform information can be added to these models to allow selection or modification of components based on these criteria.

2.4.4 Design Patterns.

Once large systems could be modeled, it became clear that the same elements were occurring over and over. These recurring elements became recognized as Design Patterns [15]. Design patterns were developed for the purpose of reuse. An example is the Adapter Design pattern. The adapter creates an intermediary abstraction between the reuse code and the system it interfaces with. The clients call a method on the adapter object, which calls the legacy component with the interface required by that component. Thus, one pattern opens reusability to many heritage components. There are other design patterns that also facilitate reuse of existing software components.

2.4.5 Simple Object Access Protocol (SOAP), Software as a Service (SAAS) and Extensible Markup Language (XML)

SOAP and SAAS began to be used widely in nonembedded systems to facilitate reuse, however, they were not frequently used in embedded systems. The use of open source software (FOSS), also popular in nonembedded systems, was limited in embedded systems due to legal issues (warranty, security) and lack of provenance.

2.5 Government Initiatives Affecting Reuse

After the Gulf War, the DoD started initiatives to leverage the large quantities of existing software products acquired during the cold war, in the hope that these products could be reused resulting in significant cost and schedule savings. At the same time, the Gulf War had made it painfully obvious that the military systems were stovepiped, making communication and sharing of information difficult, leading to multiple redundant systems and information collection activities.

2.5.1 The Software Reuse Initiative.

In 1992, the DoD published its Software Reuse Initiative, Vision and Strategy [14]. It outlined reuse principles to be used in the acquisition practices of new government-sponsored development. The initiative specified four goals:

- Improve the quality and reliability of software-intensive systems,
- Provide earlier identification and improved management of software technical risk,
- Shorten system development and maintenance time,
- Increase effective productivity through better utilization and leverage of the software industry.

The vision was "drive the DoD software community from its current "re-invent the software cycle to a process-driven, domain-specific, architecture-centric, library-based way of constructing software [14]." The initiative outlined four concepts in which families of related systems would be designed to share a common structure, and therefore allow the reuse of assets already created. Based on the 10 step strategy the initiative proposed, software reuse was expected to "bring about the cultural change necessary to make software reuse effective for the DoD."

2.5.2 The Government Acquisition Process.

Pursuant to the release of the Software Reuse Initiative, in 1993, The Acquisition Handbook required reuse in new acquisitions. Metrics were established to support the Defense Information Systems Agency (DISA)/Center for Information Management (CIM) software Reuse Program and DISA mandated that all new acquisitions have reuse requirements. Dikel drafted key elements of DOD's plan for its Software Reuse Initiative (SRI), submitted to Congress in April 1994. This initiative became part of the acquisition process [32].

2.5.3 C4ISRAF/ Department of Defense Architecture Framework (DoDAF).

The C4ISRAF is an architecture framework developed by the DoD C4ISR (Command, Control, Communications, Intelligence, Surveillance, and Reconnaissance) Architecture Working Group (AWG) to provide guidance for describing architectures. Originally published in 1996, it migrated to the DoD and became known as DoDAF in 2002.

"The C4ISR Architecture Framework is intended to ensure that the architecture descriptions developed by the Commands, Services, and Agencies are interrelatable between and among each organization's operational, systems, and technical architecture views, and are comparable and integratable across Joint and combined organizational boundaries [28]." Its intended impact was the increased development and implementation of reusable models, components, standardized interfaces to encourage more non-proprietary Commercial Off the Shelf (COTS) products and ensuring compatibility among reusable products. C4ISRAF, and then DoDAF started with four "views:" the Operational View, the System View, the Technical View and the All View. These views included 22 products, eight of which were mandatory.

In 2011, DoDAF 2.0 was released. The number of views (in 2.0 called "viewpoints") increased to eight: All Viewpoint, Capability Viewpoint, Data and Information Viewpoint, Operational Viewpoint, Project Viewpoint, Services Viewpoint, Standards Viewpoint, Systems Viewpoint. These viewpoints were now referred to as models, and were intended to be an integrated set creating a fairly comprehensive architecture. In DoDAF 2.0, none of the products were considered mandatory, the number of defined products increased to 40 (several of these 40 had sub-products, for example, the OV6 has OV6a, OV6b, OV6c). In DoDAF 2.0, the focus shifted: "DoDAF V2.0 focuses on architectural 'data,' rather than on developing individual 'products' as described in previous versions [29]."

2.5.4 Joint Technical Architecture

As computer systems used "in the field" became more sophisticated, the DoD determined that they should have access to reuse software artifacts from a number of projects. However, many of these software programs had the same names, but performed different functions. To prevent newly acquired software from overwriting existing programs, the DoD Joint Technical Architecture (JTA, updated in 2003) required that all systems adhere to open standards that facilitate interoperability, as the warfighter could be using any of several different platforms. "The standards and specifications identified in the JTA are entirely consistent with and support the DoD Standards and Acquisition Reform initiatives. The DoD standards policy recognizes the need for DoD to specify interface standards that are required for interoperability. The standards in the JTA are almost entirely performance-based interface standards. Most are commercial standards. None of the military standards require a waiver to use [33]."

The updated version in 2003 was focused on integrating systems with the Global Information Grid (GIG). "Integration of these systems into the GIG will require that they adhere to open standards that facilitate their interoperability. Transformation of DoD's capabilities, in the broadest sense, requires that existing systems are transformed in such a manner that they can share their information easily and promptly. It also requires that the GIG provide the services that allow the discovery of and collaborative use of this information for the purpose of effective and efficient business or battle-space management [34]." While many embedded systems are not part of the GIG, their products are included in information sent to the GIG. Thus, the outputs from these embedded systems needed an interface that complied with JTA. JTA also mandates platform independence (while JTA has been superceded by MOSA and is not commonly required now, it had a significant impact on reuse and many heritage programs are still contractually bound to JTA). A product developed to the standards could be used with any system that required that capability - component-based reuse was easier.

2.5.5 Modular Open Systems Approach (MOSA).

Around the same time, the Department of Defense wanted to allow payload products from one family of systems to be used in others, and accommodate products from many vendors. The hope was to eventually develop the ability to have cross-payload systems at reduced cost. MOSA was mandated as an enabler to, among other things, "enhance commonality [31] and reuse of components among systems ... [and] enhance commonality and reuse of components among systems [100]." With MOSA, proprietary interfaces were eliminated or wrapped with standardized interfaces easily accessible to other development teams. The goal was the development of models that would identify reusable or off the shelf components. MOSA eventually replaced JTA.

2.6 Success Factors for Early Reuse

Some projects had more success with reuse than others. While many efforts were reporting significant savings from reusing assets, others were not only reporting minimal success and some were even costing more than development from the beginning. To better understand this phenomenon, Applied Expertise, with contributions from WPL Laboratories, published a reuse case study [32]. The study found six principles that contributed to Nortel architecture success, namely:

- Maintaining a clear architecture vision across the enterprise
- Focusing on simplification, minimization and clarification
- Establishing a consistent and pervasive architectural rhythm
- Discovering and adapting the architecture to future customer needs, technology, competition and business goals
- Partnering and broadening relations with stakeholders
- Proactively managing software risks (and opportunities).

This case study found that the consequences of not applying the principles were severe, but that applying the principles yielded positive results. Many of these reuse strategies are still widely used, and the success factors apply to both the early strategies and the modern strategies.

2.7 Reuse in More Recent Software Development Approaches

2.7.1 Component Based Systems Engineering (CBSE)

CBSE is the process of developing trusted components and using those components as the basis of a system design. It involves the use of existing components either developed for the purpose of reuse or already in use as components or both. "Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software 'components' ([19])." CBSE recognizes that the same capabilities are required in many different situations. The components are intended to be stand alone service providers, or, encapsulated black boxes. The implementation within the component is hidden and accessed through well-defined interfaces. An expectation of CBSE was that when a system was needed, the desired components could be aggregated with little to no modification into that system.

In 1997, Goddard Space Flight Center conducted a study of the use of a domain asset library and the processes involved. The goal was to develop a configurable flight dynamics attitude support system. The result of this study was a shift from developing applications to configuring applications from reusable assets [23]. The CCT program is an example of component based systems engineering.

2.7.2 Product Lines

"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or

mission and that are developed from a common set of core assets in a prescribed way [21]." Product lines allow a development organization to reuse most of the functionality of a predefined system, while allowing some customization to nuances requested by a customer.

In 2001, the C-130J military transport aircraft program included product line reuse for an assembly line production. The C-130J project has built over 2000 aircraft for countries across the globe. Customers can select a number of variations of this aircraft. While product line reuse on this project has been beneficial, there have been some challenges. Key lessons from that project were [24]:

- Objectives and requirements must be nailed down from the beginning. It is never possible to get the requirements right the first time if the problem is of any significant degree of complexity. Requirements traceability and requirements grading are required. Conduct software product evaluations on requirements as intensely as you would review the code.

- You can never have too many simulations or laboratory resources.

- Software engineering capability maturity alone is not enough to assure the quality of an integrated system like an aircraft. Systems engineering and management capability maturity are also required.

- Driving a product by schedule is unavoidable. Be prepared to deal with it. Define all processes and measure their performance. The last process in the sequence is not necessarily the source of the problem when a schedule slips.

- Automate testing. Always plan on running a test again. Base test cases on requirements, trace test cases to those requirements, employ automated tools.

- Successful reuse requires a significant up-front cost and an effective, compelling producer/consumer model that makes it economically viable. Management must see reuse values and accept both costs and benefits.

- Measurement comes with capability maturity, but no measurements can replace the in-depth, detailed knowledge of the people on the development line. Management must journey to the (software) factory floor before they can really understand the issues

In other words, as with other strategies, having a product line does not preclude the need for solid systems engineering.

Lockheed's A2100 Bus, developed around the same time, is an example of a satellite product line. First launched in September 1996 as a geostationary earth orbit (GEO) communications bus, it is offered in several GEO configurations and will be flying in medium earth orbit (MEO) for GPS III.

2.7.3 Model Based Systems Engineering (MBSE)

MBSE involves reuse of preexisting models or prototypes. A MBSE methodology can be characterized as "the collection of related processes, methods, and tools used to support the discipline of systems engineering in a 'model-based' or 'model-driven' context....Model-based engineering (MBE) is about elevating models in the engineering process to a central and governing role in the specification, design, integration, validation, and operation of a system ([37]). Today, many systems, especially nonembedded systems, use MBSE as their development approach.

2.8 Historical Problems with Reuse

Many of the published surveys ([86], [89], [111], [109], [40]) indicate reuse success, if certain success factors are present. However, they often do not distinguish in their analyses between embedded systems and nonembedded systems. Some surveys do not include projects with mixed or negative results. In addition to the author's own experience with reuse, we have conducted a survey and found the following barriers to successful reuse:

Developer comfort with the reused products: In the early days, reuse within a program consisting of a single constellation was applied by either the developer or an associate. The developer was familiar with the reused code, its application and the culture under which it was developed. When software products were made available from outside the program, developers did not trust them or feel comfortable with them. This was cited by half of our subjects. See also McKinney [91].

Developer attitudes toward reuse: Related to the discomfort with reused products, many developers do not want to reuse software. Recent interviews with many software experts indicated that their developers prefer to create a solution over trying to fit an existing solution to their problem. When they look at the candidate reuse product, they find things that they would do differently, and decide they would rather redevelop it than adapt it. Again, over half of our respondents indicated that they or their developers were reluctant to use a product developed elsewhere.

Reuse Dictating the Product Rather than Objectives: One of the objections many customers had with large volumes of reuse is that in order to effectively import the reuse, the requirements were derived from the reused products and not from the customer objectives. Either the customer accepts a product that does not meet its objectives or reworking the requirements forces changes to the reuse products that make those products more expensive to reuse than if they had they been built from scratch. Several of our subjects reported this obstacle to successful reuse.

Architectural Mismatch: Architectural Mismatch occurs when components are costly and time consuming to integrate [47]. Often, when the reused software product is a component, the design has to be created bottom up, rather top down, defining capabilities and selecting components that best meet needed capabilities. When too many components from various sources are reused, the architecture that would be ideal for the project is hard to fit to the reused components. This can also result in a brittle final product. See also Leveson et. al. [85].

Cost: A reusable component can cost three to five times more to develop than conventional software because the component cannot make simplifying assumptions about its environment [1]. Unless the product will be reused enough to recover costs, it may not be worth the investment. See also Leveson et. al. [85].

Obsolescence: When few products are procured over a period of time, the reused products may need to be reengineered due to obsolescence before their development cost is recovered. In a NASA study [23], products had to be ported from the IBM Mainframe to UNIX workstations. Since FORmula TRANslation (FORTRAN) reuse libraries only resided on the IBM, they had to be rewritten for UNIX. Further, as software products age, and as they are modified from project to project, they become brittle and hard to maintain. Often they are written in a language that is rarely used any more and require special skills to read and update. See also Leveson et. al. [85].

Lack of adequate documentation: Many candidate reuse products have not been adequately documented through the development cycle. If documentation exists, it is often the build-to documentation rather than the as-built documentation. As a result, the reuse products do not behave as expected or to the performance levels advertised. In addition, sometimes it is harder to understand the software from the documentation than it would be to simply build new software.

Platform dependence: Many embedded software products are written and optimized to their platforms. Many of these platforms are proprietary. If a project moves to a different platform, possibly because a different vendor offers capabilities not existing in the original platform, possibly because the original vendor stops supporting the platform, the software has to be reoptimized to the new platform. In many cases, hardware is evolving faster than software, and keeping up with the hardware changes is challenging.

Lack of management/customer support: While companies and the DoD have endorsed reuse, often neither the government nor the corporation have been willing to pay for developing products for reuse. In addition, many contracts are written with limitations on sharing artifacts developed on the contract funding.

Lack of metrics to estimate and manage reuse: One of the most intractable barriers to quality reuse is the lack of metrics to estimate and manage reuse. Most projects do not have separate line items to measure and monitor reuse during any of the phases of development. This makes separating the costs and savings of reuse from new development difficult. Some projects have tried to use metrics that have been collected from nonembedded systems to estimate their embedded systems. These estimates have been less than accurate, often being off more than 100%.

Mandates: Programs cited "acquisition reform with significant unintended consequences [132]." In the mid 1990s a classified contract required heavy use of COTS, GOTS and reused software to perform a large defense mission. After significant overruns and delays, a tiger team was brought in to study the problem. The conclusion was that "when you ask a product to do what it was not designed to do, the cost in modification and maintenance exceeds any savings that may have been gotten from reuse."

Nunn McCurdy findings related to reuse problems: Senator Nunn and Congressman McCurdy introduced the 1982 Defense Authorization Act. The Nunn-McCurdy act requires that US Congress be notified if cost per unit for acquisitions exceed 15% of the original estimates and the contract be terminated if the acquisition exceeds 25% of the original estimate unless the Secretary of Defense determines that the program is necessary to national security, the cost increases are reasonable, and the management structure is adequate to constrain costs.

By March of 2011, there had been 74 breaches of this act. The Government Accountability Office (GAO) studied the reason for these breaches. "Our analysis of DoD data and SARs [sic] showed that the primary reasons for the unit cost growth that led to Nunn-McCurdy breaches were engineering and design issues, schedule issues, and quantity changes. Cost increases resulting from engineering and design issues may indicate that those programs started without adequate knowledge about their requirements and the resources needed to fulfill them [115]." Specifically, they found an inability to predict costs and benefits of reusing software artifacts. A prime example of this was the Space-Based In-

frared System (SBIRS) High program, which had breached 4 times. SBIRS High "was too immature to enter the system design and development phase and was based on faulty and overly optimistic assumptions about software reuse and productivity levels [115]." Thus, while reuse was mandated, the prediction of cost and savings was unrealistic.

2.9 The Future of Reuse in Embedded Systems

Over the past several decades, we have seen how software system development practices have changed. The industry has moved from one-time single application, stovepiped solutions to multi-use solutions that easily integrate with other systems.

Today, embedded systems are slowly reengineering their existing systems and developing new systems with frameworks and models. This is an important development, since the architecture and design phases and integration and test phases of the lifecycle are much longer and costlier than efforts in the implementation phase. Thus the architecture and design phases, and the test phases, offer more opportunities for savings. Many of these models have the capability to automatically generate code (autogen or autocode).

In a series of semistructured interviews conducted in May - July of 2013, many interviewees indicated that, while earlier, when reuse was mentioned, they had only considered code, today they are reusing many different artifacts. These artifacts include requirements, architecture, design and design products, models, documentation, and test products. When asked to rate these artifacts in terms of reuse effectiveness (defined as a series of desired outcomes), some put test products first, others put architecture first. However, all but two placed code last. One progressive embedded software subject matter expert (SME) indicated that, over time, he expected most code to be generated automatically, and the only code that would be hand written would be code that needs to be optimized to the platform.

The move to standardize platforms is having a major positive impact on software reuse. Design models are able to include the platform parameters in the model itself, and then

generate the code appropriate to the platform. Tweaking the code generator in the model allows the model to optimize the generated code to the platform, with less need for modification. Many programs today in this author's experience are reusing the tweaked autogenerator and regenerating the code for each build. The aforementioned SME predicts that, in time, the only coders will be those who specialize in optimizing very specialized code.

As more and more assets are designed to be reusable, companies are investing in reference architectures and searchable libraries. We envision these to become ontologies, which attach objectives to capabilities, the capabilities to requirements, the requirements to models, designs, test cases, and so on. These ontologies will include the documentation. A developer could select the objectives of his project, and with it select from various reusable solutions. The selected solution would be available to the developer, and the developer would only have to fill in the gaps. In many cases, large portions of these solutions will have already been tested and deployed.

So the future of software reuse in embedded systems will include:

- An increase in the reuse of architecture models and design models with code generation capabilities.
- An increase in the reuse of test products.
- A significant decrease in the reuse of code.
- An increased emphasis on parameter driven models to select from standardized platforms.
- An increased emphasis on full documentation from build to specifications to as built documents.
- Design for reuse as the norm rather than the exception, as well as reusable architectures and designs.
- An increase in reuse of test products (test drivers, test data, test seeding) and clusters of components that are already tested together.

- Infrastructures for supporting tools, techniques, methods, policies, and incentives for integrating reusable artifacts.
- A new way of thinking, with emphasis on objectives and outcomes rather than on development.

This has implications for universities, corporations and anyone who wishes to enter the software field. Universities will need to prepare students more with architecture and design skills and less with coding skills. Corporations will want to hire individuals with these skills rather than focusing on coders. The software engineering marketplace will be placing a higher premium on modeling and design and less on code.

2.10 Conclusion

Reuse approaches for over 20 years have promised lower costs for higher quality products with shorter time to deliver [4]. Customers have increasingly required reuse, management hoped to reap its promised benefits, researchers tried to demonstrate them. The Information Technology Management Reform Act of 1996, also known as the Clinger-Cohen act [26], practically mandates reuse through the Performance- And Results-Based Management initiative. However, benefits of reuse are in no way assured, even with the best of intentions. This paper tried to chronicle some of the experiences with reuse.

We saw how, in the early days of software development, developers wrote code to satisfy an immediate requirement or problem. As similar requirements or problems appeared, reuse consisted of resurrecting code the developer himself had written previously. This practice grew to creating libraries of code written by several programmers to benefit the entire team. However, there was little consideration of the practice of developing code consistently. We saw that over time, a more formal development process was implemented through structured programming. The benefit was readable code that could be more easily understood, and modified.

The next step was reusing entire components and subsystems based on similarities of new projects to the ones previously deployed. Eventually, as similar deliverable products were required, the practices evolved to include product lines. With product lines, the customers were able to select variations customized to their needs without the need to develop an entire new system.

As systems began to rely on products from outside the development teams, standards defined interfaces to facilitate integration. With the implementation of these published standards, organizations could specilize in developing commonly used components, and systems developers could procure these components rather than developing them themselves. This enabled larger systems to be developed with less effort from the system development team. Object Oriented languages enabled these components to be encapsulated. Since the interfaces to classes were well defined, outside organizations could develop their components without knowing the intricacies of the system outside of their components.

UML introduced a common way to create models of systems. The benefit of the model is that, being graphical, it was easier for many developers to understand the system and its interfaces. Design patterns allowed reuse of portions of the architecture. Using UML and design patterns, developers could import large segments of a system design without having to redo the work (sometimes with some modification, however). These became reusable assets.

Efforts by the US government and other organizations have helped make these assets more interchangeable. Focusing on architecture, models, and integration, the goal of these efforts is to encapsulate capability and eventually integrate entire systems of systems, for example the entire defense system or communication grids.

Once enough organizations were using these architectures and models to create sufficient demand, it was only natural that vendors would add code generation to their models. Nuances in the design could be included in the models and the automatically generated code would include those nuances. Many UML tools have this automatic code generation capability. As a result, requirements, architectures and models can be reused effectively.

We found that there are some key elements to successful reuse. First, reuse has to be paired with good engineering practices. Second, technical factors can be instrumental in success vs. failure, such as common platforms or environments. Third, in the defense and aerospace industry, bespoke systems may not be able to recoup reuse investment cost [85]. Fourth, a reuse mandate does not guarantee its successful execution.

While there are a large number of development approaches that reuse artifacts, legacy system reuse still dominates in embedded systems. As these heritage projects either age out or are refactored, more effort should be focused on architecture and design models. This will facilitate reuse, less reuse of code, but more reuse of the other artifacts that are more expensive to develop. We propose an ontology approach, combining objectives, requirements, architectures, models, components, source code, design artifacts, use cases, test cases and test products to make reusable products available, easy to find and easy to use (integrate).

Some studies [94] have tried to summarize reuse experiences in industry, but with limited success due to the lack of quantitative data. We recommend further research about reuse using different development approaches and the reuse of different artifacts to clarify which development approaches work best with which type of system and which artifacts bring the best return on the reuse investment. We also recommend pairing reuse with detailed measurement so that evaluating reuse success is no longer a matter of opinion.

2.11 Classifying System Types, Development Approaches, and Study Types

2.11.1 Development Approaches

Mohagheghi et al [93] define software reuse as "the systematic use of existing software assets to construct new or modified assets. Software assets in this view may be source code or executables, design templates, free standing Commercial-Off-The-Shelf (COTS) or Open Source Software (OSS) components, or entire software architectures and their components

forming a product line or product family. Knowledge may also be reused and knowledge reuse is partly reflected in the reuse of architectures, templates or processes." To simplify comparison, the Mohagheghi definition is used in this paper. However, we are specifically interested in what we call "Modern Software Reuse", that is: ad hoc reuse, component based reuse, model based reuse, product line based reuse, and ontology based reuse. We also look at combinations of these approaches. The selection of the approaches used either alone or in combination becomes, for purposes of this analysis, the reuse strategy. Table 2.1 summarizes the approaches.

Table 2.1: Development approach-specific reuse strategies

Development Approach	**Definition**	**Reuse Artifacts**
Ad Hoc/ Heritage/ Legacy	Ad hoc reuse is defined as "Unplanned and opportunistic reuse ... performed with little or no planning or commitments to produce, broker or consume assets ([88])." Legacy, or Heritage software is defined by NASA as "software products (architecture, code, requirements) written specifically for one project and then, without prior planning during its initial development, found to be useful on other projects ([96])." Grady Booch adds: "We refer to a legacy system as one for which there is a large capital investment that cannot economically or safely be abandoned([13])."	Code fragments or routines; requirements, use cases, architecture, models, drawings, test products and code.
Component	Involves the use of existing components either developed for the purpose of reuse or already in use as components or both. "Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software 'components' ([19])." Component Based Software Engineering (CBSE) is a process that aims to design and construct software systems using reusable software components. CBSE recognizes that the same capabilities are required in many different situations.	Code routines, functions, methods or other snippets contained in libraries or existing projects.

Continued on next page

Table 2.1 – continued from previous page

Development Approach	Definition	Reuse Artifacts
Model based	Reuse of preexisting models or prototypes. A MBSE methodology can be characterized as "the collection of related processes, methods, and tools used to support the discipline of systems engineering in a 'model-based' or 'model-driven' context....Model-based engineering (MBE) is about elevating models in the engineering process to a central and governing role in the specification, design, integration, validation, and operation of a system ([37])."	Architectures or design models, use cases, performance models and simulations.
Product line	Basic capabilities determined in advance and the architecture, design and code generated to map to the generic elements of the product line. Software Engineering Institute (SEI) defines a product line as "a set of software-reliant systems that share a common, managed set of features satisfying a particular market or mission area, and are built from a common set of core assets in a prescribed way ([116])." It can also be described as a family of systems sharing "a common set of core technical assets, with preplanned extensions and variations to address the needs of specific customers or market segments ([21])."	Basic requirements, core products, including architecture, requirements, use cases, components, models, test products and code reused, with some level of customization.
Ontology	A catalog of the basic capabilities that an organization is confident of delivering cross referenced to the locations of those capabilities. An ontology is a description of concepts and their relationships. "Ontology is the term referring to the shared understanding of some domains of interest, which is often conceived as a set of classes (concepts), relations, functions, axioms and instances. Ontology organizes terms with a type of hierarchy and can be drawn upon to describe the different facets with domain-specific terms...([104])." Or, "An ontology is a formal, explicit specification of a shared conceptualisation ([119])".	Retrieval information, documentation, Libraries, code clusters, requirements banks, use cases, links to models, test products and some code artifacts.
COTS/ GOTS	Commercial or Government Off The Shelf products. COTS is "Commercial-Off-The-Shelf (COTS) software components that are procured for integration into software systems ([109])".	Components, pre-integrated hardware and software, modifiable source code, specifications.

Ad Hoc Development Approach Ad hoc reuse is defined as "Unplanned and opportunistic reuse that fails to meet the full potential of reuse. It is performed with little or no planning or commitments to produce, broker or consume assets [88] ." Ad Hoc reuse is opportunistic. It is based on software products that have been previously developed and fit or approximate the need of the current work. Software was not developed with the intention of reusing it, and it is not part of a repeatable process. It is unlike "systematic reuse," when assets are developed with the expectation of being reused. While our experience indicates that it is a very common type of reuse, we found no literature based on ad hoc reuse as a reuse approach or strategy.

Product Line Based Development Approach The Software Engineering Institute (SEI) defines a product line as "a set of software-reliant systems that share a common, managed set of features satisfying a particular market or mission area, and are built from a common set of core assets in a prescribed way [116]." A software product line can also be described as a family of systems sharing "a common set of core technical assets, with preplanned extensions and variations to address the needs of specific customers or market segments [21]." In product line based reuse, the core products, including architecture, requirements, components and models, are reused, with some level of customization.

Component Based (CBSE) Development Approach Component Based Software Engineering (CBSE) aims to design and construct software systems using reusable software components [19]. CBSE recognizes that the same capabilities are required in many different situations. There is no value in developing these same capabilities from scratch multiple times. In fact, nearly every developer has, in the process of developing software, remembered doing the same routines before and incorporated his or her own or a colleague's previous work rather than going through the full development process again. CBSE formalizes this approach by defining a method to create off-the-shelf components and an accompanying, well-defined architecture. This allows software engineers to develop large systems by incorporating previously developed or existing components. It is claimed

to cause a significant reduction in development and testing time and cost [59]. It is also expected to reduce risk, in that, once validated, the components should behave the same in subsequent products as in the original. Further, there is an expectation of reduced maintenance costs associated with the upgrading of large systems.

Model Based (MBSE) Development Approach A MBSE methodology can be characterized as "the collection of related processes, methods, and tools used to support the discipline of systems engineering in a 'model-based' or 'model-driven' context....Model-based engineering (MBE) is about elevating models in the engineering process to a central and governing role in the specification, design, integration, validation, and operation of a system. For many organizations, this is a paradigm shift from traditional document-based and acquisition lifecycle model approaches, many of which follow a "pure" waterfall model of system definition, system design, and design qualification [37] ."

Ontology Based Development Approach An ontology is a description of concepts and their relationships. "Ontology is the term referring to the shared understanding of some domains of interest, which is often conceived as a set of classes (concepts), relations, functions, axioms and instances. Ontology organizes terms with a type of hierarchy and can be drawn upon to describe the different facets with domain-specific terms... [104]." In other words, an ontology is similar to a catalog, where items are cross-classified based on different sets of relationships. In software engineering, the ontology becomes a catalog of software artifacts (capabilities, requirements, services and components) that an architecture or developer can use in product development.

2.11.2 Classification of System Types

This analysis distinguishes between embedded software systems and nonembedded software systems.

Embedded Software Systems The terms embedded systems and cyber-physical systems are used interchangeably. ISO defines embedded systems as "a program which functions as part of a device. Often the software is burned into firmware instead of loaded from a storage device. It is usually a freestanding implementation rather than a hosted one with an operating system [68] ." They are further defined as "CyberPhysical Systems (CPS)... integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa [83]." Examples of embedded software include avionics, consumer electronics, motors, automobile safety systems and robotics.

Non-embedded Software Systems For lack of another definition, non-embedded software is defined as software which is not embedded, that is, software not tied to the processors or inherently integrated with the physical system. Examples of non-embedded software include web applications, desktop applications, some video games, financial systems, etc.

Some of the empirical results on reuse either include both types of systems or do not specify the types of systems. These are classified as "Both Embedded and Nonembedded" and "Unknown" respectively.

2.11.3 Empirical Study Types

To identify, catalogue and analyze empirical work assessing reuse, we follow Wohlin et al [129], Yin [131], Kitchenham [77] and Zannier et al [133]. We classified empirical studies of reuse into the following categories:

Controlled experiment Controlled experiments are experiments with random assignment of treatment to subjects, sufficient sample size, well-formulated hypotheses, control of factor level, dependent and independent variables. Because we found no examples of controlled experiments in our research, controlled experiments are not discussed further.

Quasi-Experiment In a quasi-experiment, one or more characteristics of a controlled experiment are missing. In a quasi-experiment strict experimental control and/or randomization of treatments and subject selection are missing. This is typical in industrial settings [44]. The researcher has to enumerate alternative explanations for observed effects one by one, decide which are plausible, and then use logic, design, and measurement to assess whether that might explain any observed effect [129].

Case study A case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident. In a case study all of the following exist: research questions, propositions (hypotheses), units of analysis, logic linking the data to the propositions and criteria for interpreting the findings [131]. A sister-project case study refers to comparing two almost similar projects in the same company, one with and the other without a treatment [131], such as ad hoc reuse vs model based reuse. Observational studies are either case studies or field studies. Case studies focus on a single project, while multiple projects are monitored in a field study, maybe with less depth . Case studies are observational studies with less controllable exposure to treatments, and may involve a control group or not, or being done at one time or involve analysis of historical data [77].

Survey A survey consists of structured or unstructured questions given to participants. The primary means of gathering qualitative or quantitative data in surveys are interviews or questionnaires [129]. Structured interviews (qualitative surveys) with an interview guide, investigate rather open and qualitative research questions with some generalization potential. Quantitative surveys with a questionnaire, contain mostly closed questions. Typical ways to fill in a questionnaire are by paper copy via post or possibly fax, by phone or site interviews, and by email or web [70]. For purposes of this paper, surveys are questions asked of individuals, such as engineers, rather than companies, the surveys of companies in industry are included under reviews of industry practice.

Review of Industry Practice Similar to a survey, a review of industry practice consists of discovering and analyzing the ways different companies in the industry perform their tasks. Often they share the same data collection techniques with a survey, but the questions focus on company practice.

Meta Analysis A meta analysis consists of analyzing multiple studies on the topic in question. Meta analysis covers a range of methods to generalize and compare results of a group of studies.

Experience Report An experience report is similar to a case study, but it does not have the same level of controls or measures. It is retrospective, generally lacks propositions, may not answer how or why phenomena occured, and often includes lessons learned [133]. In this paper we combine example applications with experience reports because most papers had features of both types of studies. An example application consists of "authors describing an application and providing an example to assist in the description, but the example is 'used to validate' or 'evaluate' as far as the authors suggest [133]" but without the rigor of a formal case study.

Expert Opinion An expert opinion provides some qualitative, textual, opinion-oriented evaluation. It is "based on theory, laboratory research or consensus [77]." These expert opinions assess processes, strategies, approaches, theoretical models, policies, curriculum or technology, that may or may not allude to full-scale evaluation or empirical studies. Often such articles are based on experience, observations, and ideas proposed by the author(s).

Chapter 3

Review of Existing Literature

There is a debate in the aerospace industry whether lessons from reuse successes and failures in nonembedded software can be applied to embedded software. This chapter analyzes and compares reuse success and failures in embedded versus nonembedded systems. A survey of the literature identifies empirical studies of reuse that can be used to compare reuse outcomes in embedded versus nonembedded systems. Reuse outcomes include amount of reuse, effort, quality, performance and overall success. We also differentiate between types of development approaches to determine whether and how they influence reuse success or failure. In particular, for some development approaches, quality improvements and effort reduction are low for embedded systems. This is particularly relevant to the Aerospace industry as it has been subject to reuse mandates for its many embedded systems.

3.1 Introduction to Review of Existing Literature

Reuse supposedly reduces development time and errors. "If a software package has been executing error-free in the field for an extended period, under widely varying, perhaps stressful, operating conditions, and it is then applied to a new situation, one strongly expects that it should work error free in this new situation [4]" and "In theory, reuse can lower development cost, increase productivity, improve maintainability, boost quality, reduce risk, shorten life cycle time, lower training costs, and achieve better software in-

teroperability [101]." The Aerospace Industry was an early advocate of reuse. Chapter 8 of Anderson and Dorfman [4] discusses reuse in aerospace, including potential savings in quality, cost and productivity. In July of 1992, the DoD released the DoD Reuse Initiative: Vision and Strategy [14]. The government invested heavily in reuse, e.g. the Control Channel Toolkit (CCT) in 1997 and Global Broadcasting Service (GBS) beginning in 1998. Hence, one would expect large scale planned reuse. Many government requests for proposals contain a requirement for quantifying expected savings from reuse.

However, from the beginning, there has been a debate on reuse success. In Aerospace, not all reuse experiences have been as successful as expected. Some major projects experienced large overruns in time, budget, as well as inferior performance, at least in part, due to the wide gap between reuse expectations and reuse outcomes [115, 132]. This seemed to be especially the case for embedded systems. In many US government customer shops, reuse became a red flag when awarding contracts.

Many engineers believed that one of the root causes of the disconnect between reuse expectations and reuse realization was that too often estimated savings from reuse came from projects that differed from the target project. In particular, the reuse estimates for nonembedded systems were applied to embedded systems. There was an ongoing debate as to whether embedded systems and nonembedded systems were similar as far as reuse was concerned. Many systems and software engineers, especially those who worked on embedded systems, claimed that reuse could not be successfully used in their systems because the code was optimized to particular processors which may not be used in the new project. Many of the embedded systems software engineers claimed that reusing software was more costly than building the system from scratch. Further, many claimed that, in particular, trying to use model based development (derived from the Initiative) was more costly than using other development approaches. Our major motivation was to investigate whether empirical studies exist that either support or contradict these opinions.

Our research questions are:

- Are embedded systems different with respect to reuse?
- Do embedded systems employ different development approaches?
- Does the development approach have an impact on reuse outcomes?
- What types of empirical studies exist that analyze and/or compare reuse in embedded and nonembedded systems? Given that empirical studies can vary greatly in their rigor, this should indicate how "hard" the evidence is.
- To what extent is there solid quantitative data paired with appropriate analysis?
- Are there studies that either deal with aerospace projects or can reasonably be generalized for this domain?
- What are the limitations of the current empirical evidence related to reuse?

In 2007, Mohagheghi and Conradi [93] conducted a survey assessing reuse in industrial settings. They studied the effects of software reuse in industrial contexts by analyzing peer reviewed journals and major conferences between 1994 and 2005. Their paper's guiding question was "To what extent do we have evidence that software reuse leads to significant quality, productivity or economic benefits in industry?" Mohagheghi and Conradi [93] is a major step forward in identifying and measuring reuse effectiveness. Unfortunately, their work does not distinguish embedded vs non embedded systems. By contrast, this chapter

- Compares reuse effectiveness in embedded vs nonembedded systems
- Compares reuse effectiveness for different development strategies

3.2 Review Process and Inclusion Criteria

The search considered studies published in peer-reviewed journals and conferences, industry forums such as SEI, industry seminars, symposia and conferences, and industry

and government-funded studies. Industry sources were especially useful for Product Line development, since academic sources rarely have the need or ability to develop a product line for evaluation purposes. Additional sources were monographs and technical reports (for example, Hall et al [60]).

We searched the ACM digital library and IEEE Xplore, Empirical Software Engineering Journal, Journal of Systems and Software, Journal of Information Science, MIS Quarterly (MISQ), IEEE Transactions of Software Engineering (TSE), IT Professional, ACM Computing Surveys (CSUR), the Journal of Research and Practice in Information Technology, Springer Verlag, and Google Scholar. Keywords included "reuse," "reuse benefits," "reuse case study," "reuse empirical study," "product line," "component based," and "model-based."

The articles were filtered by relevance based on titles. From relevance of titles alone, we reduced the search to about 400. Reading the abstract and conclusion reduced the number to 126, which were finally cut to 55 after reading the full article.

Once we had selected an initial set of articles, we considered works cited by the 55 papers, adding about 12. We also included newer works by researchers whose papers were relevant, adding 7. In addition, we added from grey literature articles such as case studies in industry published by the Software Engineering Institute (SEI). The final set of articles about development approaches and strategies came to 83. (This does not include basic background information about software development, research methods, etc.) Due to our primary interest on the impact of reuse mandates since the Software Reuse Initiative, we only considered papers published between 1992 and 2013.

The papers were classified by study type (Section 4.1, 4.2, 4.4), system type (Section 4.1, 4.2) and development approach (Section 4.3). Once the papers were classified, it became clear that many reported on a particular reuse strategy or method, but were not discussing the value of reuse per se, nor were they performing a comparison against other methods or between types of systems. Because these did not add to the analysis, a threshold was established requiring that at least 20 per cent of the paper be devoted

to a discussion of the merits of reuse itself or comparison with other methods. While [124] find that only papers that devote at least 30% of their content to empirical results contain adequate experimentation, we determined that setting the threshold this high would exclude important data. This criterion resulted in the removal of sixteen papers since they discussed (similar to Henninger [62]) empirical results only peripherally. The removed papers, ([12,27,54,55,63,64,66,70,73,78,82,104,126,127,134,135]) were from the experience report and expert opinion categories. We also excluded textbooks (e.g. [74]), since their major purpose is to teach a methodology rather than to evaluate reuse success.

Finally, in 24 papers we could not determine whether the systems were embedded or nonembedded. This was disappointing, because highly regarded papers about reuse did not identify the system type, and thus could not be used in our analysis. We had to exclude the following papers: [11,26,30,36,38,41,45,59,70–72,76,92,93,97,97,106–108,110,111,117,130], and [9]). This was particularly unfortunate, since they contain some highly regarded work. The final paper count for analysis was 43.

Seventeen articles evaluated reuse in embedded systems only, 17 evaluated reuse in nonembedded systems only, and nine dealt with empirical results for reuse in both embedded and nonembedded software systems. We found 10 discussions of case studies, one quasi-experiment, three surveys, two meta-analyses, four reviews of practice, nine expert opinions, and 14 experience reports.

While some papers discussed various development strategies, and embedded systems and nonembedded systems, none compared development approaches with each other and none compared reuse in embedded systems against reuse in nonembedded systems. This study compares reuse outcomes from studies using different development strategies, and outcomes from studies using embedded and nonembedded systems. In the studies that covered both embedded and nonembedded systems, we collected the data from each for comparison.

3.3 Reuse and Development Approaches for Embedded vs. Nonembedded Systems

We classified papers by type of system (embedded and nonembedded), development approach, and type of empirical research. Some empirical studies covered combinations of two or more approaches. While the academic definitions of some approaches subsume other approaches, it was not clear that this was happening. One argument against is that embedded systems tend to include performance and reliability models, MATLAB models etc. Further, over time, some parts of the system may have switched development approaches. Additionally, some studies reported on multiple projects. If their development approaches were not the same, the study was classified in more than one category. Therefore, we counted each development approach separately. An example of multiple classification is [39]. We distinguished studies by whether they provided qualitative data (e.g. the paper reported success as high, medium, low or reuse outcomes as better, worse, or reuse satisfaction as satisfied, dissatisfied) or quantitative data (e.g. per cent improvements, or r-values, p-values in statistical analysis results).

3.3.1 Software Reuse in Embedded Systems

Seventeen empirical studies covered reuse in embedded software. These included empirical studies of reuse in industry and at government agencies. When more than one reuse development approach was involved, the study was counted in multiple categories. Table 3.1 shows the data by development approach and study type.

Of the four case studies covering development strategies in embedded systems exclusively, one was a combination of ontology and model based reuse [102], two were product line reuse [84], [58], and one did not specify development approach [10]. The empirical study evaluating the combination of ontology and model based reuse involved an Ericsson product. It was based on interviews of Ericsson senior modelers. The particular object of study was a subproject within the company focused on developing embedded software

Table 3.1: Empirical Studies of Embedded Systems Studies Reuse by Development Approach

Embedded Systems	Case Study	Quasi-Experiment	Survey	Review of Practice	Meta-Analysis	Experience Report	Expert Opinion	Total
Ontology	1							1
Product Line	2	1				2		5
Model Based	1	1				3	2	7
Component Based						2		2
Unspecified	1					1	2	4

* Some studies included more than one development approach due to reporting on multiple projects, hence 19, rather than 17 studies.
** There were no surveys, reviews of practice or meta-analyses that dealt with reuse in embedded systems.

for part of a mobile-communications-network product. The study identified 26 areas for improvement in modeling content, activities and management for large projects [102].

The one quasi-experiment on reuse in embedded systems studied both component-based and model-based development approaches [17]. They developed a method (MARMOT) to analyze performance of model-based, component-oriented development in a small system. The system was a control system for an exterior car mirror.

Four expert opinion papers covered reuse in embedded software. Dos Santos and Cunha discussed an embedded satellite system [35]. One discussed modeling interacting hybrid systems [2]. The other two examined reuse trends in embedded software in real time signal processing systems: one considered application and architecture trends, the other considered design technologies [53,103]. These articles dealt with architecting and designing a very large scale integrated (VLSI) chip.

Eight experience reports evaluated reuse in embedded systems. For example, one was a DARPA analysis from model-based designers working with robotics, including tools and techniques ([118]). The combination product line/component based reuse study involved digital audio and video projects [75]. They developed a process for platform development, found core assets in the digital AV domain and legacy assets, designed a common architecture and reported their experience with this approach. The report on component based reuse involved field devices, such as temperature, pressure and flow sensors, actuators and

positioners, in other words, small, real-time embedded systems ([128]). Two of the studies on model based reuse were about the TechSat21 program, and appear to be the same study [114, 121]. They created an agent-based software architecture for autonomous distributed systems. Through Matlab, they were able to model clusters in a way that enabled operators to command the cluster as a virtual satellite by decomposing goals into specific satellite commands.

3.3.2 Software Reuse in Nonembedded Systems

Seventeen studies involved reuse in nonembedded software in industry and at government agencies. Table 3.2 shows the studies by development approach and study type.

Table 3.2: Empirical Studies of Nonembedded Systems Studies Reuse by Development Approach

Nonmbedded Systems	Case Study	Quasi-Experiment	Survey	Review of Practice	Meta-Analysis	Experience Report	Expert Opinion	Total
Ontology				1		1	1	3
Product Line	1						1	2
Model Based	2					1	1	4
Component Based	3		2			1		6
Unspecified	2						3	5

* Some studies included more than one development approach due to reporting on multiple projects, hence 20, rather than 17 studies.
** There were no quasi-experiments or meta-analyses that dealt with reuse in nonembedded systems.

Seven reuse case studies addressed reuse in nonembedded software (since one covered two development strategies, it is reported twice). Two studies addressed model based reuse, two studies addressed component based reuse, one studied a combination of product line and component based reuse. Two did not specify the development approach. The following are examples of case studies in nonembedded systems. One study of a model based approach involved reducing interface incompatibilities via feature models [80]. The study on a combination of product line and component based reuse involved large telecommunications products. The components were built in house and shared across product lines. Standardized processes and architectures enabled the reuse [94]. One study on component

based reuse reported on an industrial case study in a large Norwegian Oil and Gas company, involving a reused Java class framework and two applications. It analyzes reasons for differences in defect profiles [57]. The other involved research performed by the National Aeronautics and Space Administration (NASA) Earth Science Data Systems (ESDS) Software Reuse Working Group (WG) that was "established in 2004 to promote the reuse of software and related artifacts among members of the ESDS data product and software development community." Artifacts were shared via a web portal [50].

There were two surveys that specifically addressed a component-based development approach in nonembedded software [86]. A questionnaire (answered by 26 developers) covered areas dealing with the practice and challenges for what they called "development with reuse in the IT industry." They studied company reuse levels and factors leading to reuse of in-house components. The other addressed success factors in reuse investment [108].

One review of practice studied the use of ontologies [20], including ontology mapping categories and their characteristics. This included four different ways of merging ontologies into a single ontology from multiple sources.

Four expert opinion papers discussed nonembedded systems. One expert opinion paper on development strategies covered ontology, product line and model based reuse [39]. It analysed developing an ontology of models into a product line in the insurance domain. One discussed the difficulties of architectural mismatch ([48]). One investigated reuse libraries and their contributions to reuse ([92]). One discussed success factors across business domains [107].

Three experience reports addressed reuse in nonembedded systems. The ontology experience report addressed processes and product development using an ontology for the budget domain of the Public Sector in Australia [15]. The experience report on model based reuse covered domain-specific modeling [16]. It discussed stock trading tasks such as buying and selling stock, and creating user account details. The experience report on component based reuse discussed the SMC satellite ground system framework [120].

Clearly, compared to embedded systems, we find more case studies, most of which are dealing with CBSE. All empirical studies relating to ontologies are qualitative (review of practice,, expert opinion and experience report), maybe reflecting that ontologies are still somewhat novel in industry. Reuse in product line development shows predominantly qualitative data as well (3:1). Even reuse in model-based development shows more qualitative (expert opinion and experience report) than quantitative analysis (case study). Reuse in CBSE has more quantitative results in its case studies. This is matched by 4 qualitative studies (survey, review of practice, experience reports). Empirical studies of reuse that do not identify the development approach are also predominantly qualitative (4:1) with three expert opinion studies compared to one case study and a survey. Clearly, the goal to analyze and compare reuse outcomes in embedded vx nonembedded systems for different development approaches would have been helped greatly with more "hard" data.

3.3.3 Software Reuse in Embedded and Nonembedded Systems

Nine studies covered development strategies in both embedded and nonembedded systems. Table 3.3 shows the type of data collected by development approach and study type.

Table 3.3: Empirical Studies of Embedded and Nonembedded Systems Studies Reuse by Development Approach

Both Embedded and Nonembedded Systems	Case Study	Quasi-Experiment	Survey	Review of Practice	Meta-Analysis	Experience Report	Expert Opinion	Total
Ontology								
Product Line			1			2	1	4
Model Based	1		1				1	3
Component Based	1							1
Unspecified				1	2		1	4

* Some studies included more than one development approach due to reporting on multiple projects, hence 12, rather than 10 studies..
** There were no quasi-experiments that dealt with reuse in both types of systems.

Two case studies examined both embedded and nonembedded software systems. It was is a side-by-side comparison of two very different systems, a pump controller vs web mail. ([125]) It did not specifically address reuse characteristics that may be different between

embedded vs nonembedded system . The other case study examined data from two reuse projects at Hewlett Packard (HP), one an embedded system that developed, enhanced and maintained firmware for printers and plotters. The other is a nonembedded system that produces large application software for manufacturing resource planning [87].

The reuse survey covering both embedded and nonembedded systems addressed product line/model based reuse for both types of systems. Its purpose was to identify some of the key factors in adopting or running a company-wide software reuse program. The key factors were derived from information gained from structured interviews dealing with reuse practices, based on projects for the introduction of reuse in European companies. Twenty four projects from 1994 to 1997 were analyzed. The projects were from both large and small companies, a variety of business domains, using both object-oriented and procedural development [109].

The two meta-analyses studied reuse (without clearly specifying the development approach). One was a study of reuse literature published between 1994 to 2005 covering benefits derived from software developed for reuse and software developed with reuse [93]. The other discussed factors that could lead to reuse success [43].

Two expert opinion papers discussed reuse in both embedded and nonembedded software systems. One expert opinion dealt with adding a model based reuse to a product line reuse development approach, thus developing and documenting the product via a model [79]. It discussed the creation of a model that would allow firms to develop related lines on a common product platform. Thus the purpose of this study was model building, rather than a development approach assessment.

There were two experience reports. One report on product line reuse compared two case studies producing very different products (factory automation and medical information) using the configurable software product family approach [106].

Similar to the previous two categories of analysis, qualitative studies outnumber quantitative ones (6:3). More recent development approaches either have no empirical studies (ontology) or the studies are more in the expert opinion or experience categories (product

line). Reuse in Model Based development shows quantitative data through a case study and a survey. The most extensive analysis of reuse with meta-analysis was performed on systems for which the development approach was not specified. This does not help in evaluating where and how the development approach may or may not have contributed to success.

As can be expected, empirical studies of reuse with specific development approaches start with experience reports and expert opinions, progressing to more in-depth case studies, quasi-experiments and finally meta-analysis over time.

3.3.4 Comparing Study Types

In total, seven types of empirical studies were represented in our review. Of these, three were of the more rigorous type (case study, quasi-experiment, survey), two were a less rigorous type (review of practice and meta-analysis) and two were the least rigorous (expert opinion and experience report). When we look at the numbers of papers in these catagories, five studies of embedded systems, nine studies of nonembedded systems and three of both embedded and nonembedded systems were of the most rigorous types. No studies of embedded systems, one of nonembedded systems and two studies of both embedded and nonembedded systems were of the less rigorous types. Twelve studies of embedded systems, seven studies of nonembedded systems and four studies of both embedded and nonembedded systems were of the least rigorous type of study.

Case studies, quasi-experiments, surveys, reviews of practice and meta-analyses provided quantitative data. The expert opinion and experience reports offered mostly qualitative information. One experience report offered quantitative data.

Reuse with an ontology approach was only evaluated once with a case study including only qualitative data. Reuse in product line reported quantitative data in two studies (a case study and a quasi-experiment), the rest included qualitative data (a case study and experience reports). Reuse in model based development had quantitative data in three studies (a case study, a quasi-experiment and an experience report), but there were four

less rigorous expert opinions and experience reports (all with qualitative data). Reuse in CBSE only reported qualitative data in experience reports. The studies that did not identify the development approach also were predominantly qualitative.

3.4 Metrics Reported

Next we turn to the papers' reporting of metrics. The types of metrics included size, reuse levels, quality, effort, performance and programmatic (such as staff, institutionalized process, or schedule). We noticed that, while all of the metrics reported fit into these categories, the way the metrics were collected and reported differed. For example, size could mean the size of the project, the software size, the model size, the size of the system, the size of the software staff or the size of the project. Reuse level could refer to reused elements (total number or per cent of the project), frequency in which certain components were reused, phase of reused assets, and reused requirements. Quality referred to defects, faults, severity of defects, reliability, and aspects of errors. Effort was reported in terms of phase (development, design, simulation), productivity, and rework. Programmatic reported schedule, staff, process and time to market.

Table 3.4 shows metrics reported in studies of reuse of embedded systems. They include size, amount of reuse, quality, effort and performance. Only one study reported performance metrics, which was somewhat surprising, since many embedded systems have to meet real-time performance requirements or deal with limited storage capacity. Reuse studies on embedded systems did not report metrics related to process or programmatics. All size metrics were ratio, as were quality and effort. Reuse metrics were ratio except for needs vs needs met, which was ordinal.

Table 3.4: Metrics Used in Studies of Reuse of Embedded Software Systems [1]

Attribute	Definition	Scale	Explanation	Source
Size				
Model Size - Absolute	"The numbers of elements: the number of classes, the number of use cases, the number of sequence diagrams, or the number of classes in a diagram."	Ratio	"Absolute Size 'Length' can be measured by source code, it is organized as a sequence of characters and lines. UML models are not sequences and there exists no meaningful notion of length. So we replace 'length' by 'absolute size'. Metrics that measure a model's absolute size are the numbers of elements."	[17], [81]
Model Size - Relative	"Ratios between absolute size metrics, such as number of sequence Diagrams, number of objects; number of use cases, number of vlasses; number of state charts/ number of classes"	Ratio	"These metrics enable to compare the relative size (or proportions) of different models with each other and they give an indication about the completeness of models."	[17], [81]
System size.	KBytes of the binary code	Ratio	Executable	[17],
Size	Terminal semicolons, Delivered source instructions	Ratio	Source Code	[10]
Reuse				
The amount of reused elements	"The proportion of the system which can be reused without any changes or with small adaptations."	Ratio	Measures taken at model and code level.	[17],
Reusable Requirements	Reusable requirement unit	Ratio	Any prominent and distinctive concepts or characteristics that are visible to various stakeholders.	[84]
Needs vs needs met	Direct needs expressed by the informants, indirect needs inferred from descriptions of situations or problems	Ordinal	As defined by surveyed engineers	[102]
Quality				
Defect density	Defects per 100 LOC	Ratio	"Collected via inspection and testing activities."	[17]
Reliability	Reliability of the software		Assumed to be 100per cent	[114]
Effort				

Continued on next page

[1] *Performance metrics reported only in embedded systems.

Table 3.4 – continued from previous page

Attribute	Definition	Scale	Explanation	Source
Development effort	Development time (hours)	Ratio	Collected by daily effort sheets.	[17]
Simulation Effort	Person weeks	Ratio	Time to develop simulation environment	[85]
Performance				
Performance	On board fuel consumption and time to perform simulation		"Because time for communication has greatest impact, only communication time included."	[114]
Computation	Current CPU workload and total workload	Ratio	Per cent of maximum computational rate (comp/sec) or per cent of CPU time dedicated to a task	[114]

Table 3.5 shows metrics reported in studies of reuse of nonembedded systems. Except for performance metrics, the same categories of outcomes are reported, but there is a larger variety of measures for several of the categories. We notice, for example, that nonembedded systems report the frequency with which assets were reused as a measure of reuse, not reported by embedded systems. In quality, nonembedded systems report severity of defects, changes to software products, and component understanding, which are also not reported by embedded systems. Nonembedded systems report on productivity, an effort metric not mentioned by embedded systems.

Nonembedded systems also had a greater division of the metric scale. While all of the size metrics were ratio, in reuse level four were ratio and three were ordinal. In quality, four were ratio to two ordinal, and in effort three were ratio and one ordinal. This limited the types of analysis that we could perform.

Table 3.5: Metrics Used in Studies of Reuse of Nonembedded Software Systems [2]

Attribute	Definition	Scale	Explanation	Source
Size				
Software Size	Non Commented Source Lines of Code	Ratio	Source Code	[57]
Module Size	KLOC	Ratio	Source Code	[69]
Size	KSLOC	Ratio	Source Code	[94]
Reuse				

Continued on next page

[2]*Performance metrics reported only in embedded systems.

Table 3.5 – continued from previous page

Attribute	Definition	Scale	Explanation	Source
Reuse Level	Ratio of different lower level items reused in higher level items to total lower level items used.	Ratio	RL is based on counting item types rather than item tokens.	[44]
	Developers assessments of reuse levels	Ordinal	Developers assessments of reuse levels	[86]
Reuse Frequency	Frequency of references to reused items.	Ratio	Percentage of references to lower level items reused verbatim inside a higher level item versus the total number of references.	[44]
	Frequency of module reuse	Ratio	Ratio of the yearly sum of reuse frequencies to number of modules stored in the library	[69]
Active Module Ratio	Ratio of active modules	Ratio	Ratio of number of modules reused at least once each year to number of modules stored in the library	[69]
Component requirements (re)negotiation	Developers Assessments of Component related requirements (re)negotiation	Ordinal		[86]
Repository Value	Developers assessments of Value of component repository	Ordinal		[86]
Quality				
Defect Density	The NSLOC of each system divided by the number of defects based on trouble reports	Ratio		[57]
Fault density	The number of faults divided by the software size	Ratio	An error correction may affect more than one module. Each module affected is counted as having a fault.	[94]
Severity	The number of defects of different severities divided by the NSLOC	Ratio		[57]
Number of module deltas	A change to a software work product such as code	Ratio	Either an enhancement or a repair, and since they correlate well with faults, deltas are sometimes used for estimating error rates	[44]
Component Understanding	Developers assessments of component understanding	Ordinal		[86]

Continued on next page

Table 3.5 – continued from previous page

Attribute	Definition	Scale	Explanation	Source
Quality Attributes of Components	Developers assessments of component attribute definitions	Ordinal		[86], [44]
Effort				
Time to Develop	Staff Months	Ratio	Staff Months	
Productivity	Number of NCSLs produced per person day	Ratio	Relies on the effectiveness of NCSL as a measure of product size.	[44]
	Effort in person days spent per module	Ratio	Effort to develop unit of compilation and deployment. A higher Effort/ Module, means lower productivity	[44]
Developer Attitudes	How developers felt reuse process affected productivity and quality	Ordinal		[56]

Table 3.6 shows metrics reported in studies of reuse in both embedded and nonembedded systems. Unlike studies of only embedded or only nonembedded systems, we see metrics related to programmatics. Also unlike studies of only embedded or only nonembedded systems, the only reuse metric was the phase of reuse, and only this category measured rework.

Table 3.6: Metrics Used in Studies of Reuse of Both Embedded and Nonembedded Systems[3]

Attribute	Definition	Scale	Explanation	Source
Size				
Team size.	Persons	Ratio	Number of persons on the reuse team	[109]
Program Size	KSLOC	Ratio	Source Code	[109]
Reuse				
Phase of Reuse	Phase in which artifacts are reused	Nominal		[109]
Quality				
Defect Density	Defects per KSLOC	Ratio		[101]
Error Source	Source of error	Nominal	Where did the error originate	[93]

Continued on next page

[3]*There were no performance metrics reported in studies of both embedded and nonembedded systems.

Table 3.6 – continued from previous page

Attribute	Definition	Scale	Explanation	Source
Type of Error	Error Type	Nominal	Level of severity of error	[93]
Error Slippage	Error Slippage	Ratio	Error Slippage from Unit Test	[93]
Effort				
Development Effort	Effort in person hours	Ratio	Effort per module, asset or product in person hours, days or months	[93]
Design Effort	Per cent of total effort	Ratio	Per cent of development time spent in design	[93]
Effort	Person Months	Ratio		[93]
Productivity	LOC/time	Ratio	Apparent and actual LOC per time unit, size of application divided by development effort	[93]
Rework	Person Months	Ratio	Effort spent in isolating and correcting problems, difficulty in error isolation or correction	[93]
Programmatic				
Schedule	Months	Ratio	Calendar time to complete	[101]
Time to Market	Per Cent	Ratio	Reduction in Time to Market	[93]
	Months	Ratio	Months to Deliver	[109]
Staff	Persons	Ratio	Overall staff size, software staff size	[95]
	Years	Ratio	Overall staff experience, software staff experience	[95]
Process	Software Process	Nominal	Presence of reuse process integrated into organization software development process	[109]

Table 3.7 summerizes the commonalities and differences of metrics collected for embedded systems and nonembedded systems as well as the metrics collected when both types of system were studied. It shows that in the subattributes, there was little commonality in the metrics reported. It is only at the generalized outcome level that the reported metrics become comparable.

In summary, metrics varied widely, ranging from nominal to ratio level of measurement, and metrics for the same attribute varied quite a bit among studies. This makes direct comparison difficult.

Table 3.7: Comparing Metrics Used for Embedded Systems, Nonembedded Systems and Both Types of Systems[4]

Attribute	**Embedded**	**Nonembedded**	**Both Types**
Size			
Software Size		X	
Model Size - Absolute	X	X	
Model Size - Relative	X		
System size.	X		
Size	X	X	
Team size			X
Program size			X
Reuse			
The amount of reused elements	X		
Reusable Requirements	X		
Needs vs needs met	X		
Reuse Level		X	
Reuse Frequency		X	
Active Module Ratio		X	
Component Requirements R(e)negotiation		X	
Repository Value		X	
Phase of Reuse			X
Quality			
Defect density	X	X	X
Fault density		X	
Severity		X	
Number of Module Deltas		X	
Reliability	X		
Component Understanding		X	
Quality Attributes of Components		X	
Error Source			X
Type of Error			X
Error Slippage			X
Effort			
Development effort	X		X
Design Effort			X
Simulation Effort	X		
Time to Develop		X	
Productivity			
Effort			X
Rework			X
Developer Attitudes		X	
Performance			
Performance	X		
Computation	X		

Continued on next page

[4]*Note that the types of attributes studied were more programmatic when both embedded and nonembedded systems were studied.

Table 3.7 – continued from previous page

Attribute	Embedded	Nonembedded	Both Types
Programmatic			
Schedule			X
Time to Market			X
Staff			X
Process			X

3.5 Analysis of Outcomes

Unlike other studies that analyzed papers reporting on multiple studies, but reported results as a single data point per paper (e.g. [93]), we scored each project individually. For example, one study included 27 different projects with different results for different projects ([109]). These are scored as 27 individual data points. When a study reported on reuse in both embedded and nonembedded systems, we included the individual projects in both catagories, as appropriate. The remainder of this chapter reports on analyzing empirical evidence of reuse by project rather than by paper (or study). in both embedded and nonembedded systems ([109]).

Comparing Development Approaches. Table 3.8 shows how many projects used a particular development approach. In comparison, there were fewer studies investigating reuse with a product-line approach for embedded systems than for nonembedded systems (20 vs 23). Embedded system reuse was studied for model-based development approaches 19 times, compared to 24 times for nonembedded systems. CBSE was studied more frequently for reuse in nonembedded systems than embedded systems (14 vs 7). It appears that across the board, reuse was studied in fewer projects in embedded systems than nonembedded systems.

Table 3.8: Number of Projects by Development Type

	Ontology	Product Line	Model Based	Component Based	Unspecified	Total
Embedded	0	20	19	7	10	56
Nonembedded	3	23	24	14	15	79
Total	3	43	43	21	25	135

Comparing software size. We measured software size as in ([95]): (size of the software project on which reuse was applied): Small = less than 10 KLOC and 10 person-months effort; Medium = 10-100 KLOC and 10-100 person-months; Large = more than 100 KLOC, more than 100 person months. Table 3.9 compares the sizes of the embedded systems projects by development approach. For each development approach,"NR" states the number of such projects reported and % compares the percentage of projects in each size category. Table 3.10 does the same for nonembedded systems. Where possible, we also used synonyms of the sizes reported in Table 3.7. Because many empirical studies did not report system size, the numbers in the tables do not add up to the total number of projects.

Table 3.9: Size of Embedded Systems

Embedded[a]	Product Line		Model Based		Component Based		Unspecified Reuse	
Project Sizes	NR	%	NR	%	NR	%	NR	%
Large	4	40	2	14	2	67		
Medium	6	50	3	21				
Small	1	10	9	65	1	33	2	100

a]Note many papers did not report size

Table 3.10: Size of Nonembedded Systems

Nonembedded[a]	Product Line		Model Based		Component Based		Unspecified Reuse	
Project Sizes	NR	%	NR	%	NR	%	NR	%
Large	1	50			6	60		
Medium								
Small	1	50			4	40		

a]Note many papers did not report size

Empirical studies of both Product Line and Component Based development strategies use a larger proportion of medium and large systems, while studies of model-based and unspecified development strategies tend to study small systems. Given that embedded systems cover small, medium and large systems, this represents a reasonable cross section.

Comparing outcomes. Let us turn to outcomes next.

Given the diverse ways in which various system attributes and reuse variables were measured, as well as the lack of effect size in most studies, it was not possible to perform a meta-analysis that could quantitatively assess and compare similarities and differences between outcomes. In an effort to quantitify and analyze the data, we include as "characteristics of events" the concepts of "better or worse." "Better" meant that the reuse outcome under study was considered to be an improvement over not reusing. Conversely, "worse" meant that the reuse outcome had negative results. We use the term "mixed" when there was in improvement in some aspects of the outcome and other aspects of the outcome were negative, or when there was no noticeable difference between reusing and not reusing assets. Thus we are able to assign an ordinal measure to outcomes, i.e. 1 for better outcomes, -1 for worse outcomes and 0 for either no change in outcomes or mixed results. We use the same outcome categories as in Table 3.4. Tables 3.11 and 3.12 list development approach as major column headers like Tables 3.9 and 3.10. For each development approach we list number of projects for embedded (E) and nonembedded (N) systems. Rows list outcome scores within each outcome category. While three nonembedded systems projects report using an ontology approach, there were no embedded system projects using an ontology. Hence we could not compare them and left them out of Tables 3.11 and 3.12.

A relatively large number of papers only reported very high level results rather than results for specific attributes of reuse outcomes. These are scored similarly and shown in Tables 3.11 and 3.12 under "general." The "general" category reflects the developers' reported overall reuse experience.

Embedded Systems

For *reuse level* in embedded systems, most of the studies reported positive reuse results for product line, model based, and component based development. This was not the case for the two studies that did not specify development approach. One of them reported mixed, the other negative results. For product line reuse in embedded software systems, we identified three reasons for reuse failure: complex reuse repository, lack of management commitment and changing of the hardware. With a model based development approach, the failure may have been due to lack of experience in modeling. Under unspecified development approach, we found that for embedded systems, the results were no better and possibly worse than if no reuse had been employed. However there is no explanation in the studies as to why that would be the case.

While *effort* was reduced in all projects with product line development that reported effort, this was not the case for model-based development, where the majority of studies reported negative (7) or mixed (6) results. Each of the three projects using component based development reported a different outcome with regard to effort. Finally, when development approach was not specified, two projects reported positive, one a mixed outcome. It appears that while the majority of projects report positive outcomes with regard to effort, some development approaches show more mixed and negative results. This would indicate that the type of development approach matters.

Improvements for *quality* do not always materialize, as 8 negative results indicate. They are related to model-based and unspecified development approaches. By contrast, all four projects (that reported quality outcomes) that used a product line approach report positive outcomes for quality, as opposed to 2 for model based development (compared to five negative and six mixed results) and only 1 for component based development reported a positive outcome. Again, development approach matters.

Projects that only report *overall reuse success* are generally favorable (19), but again, model based development projects reported one negative result, six mixed results and only two positive ones. In the model based reuse projects that reported mixed results, while

problems were identified, only the factors leading to the success of the successful projects were discussed. In the model based reuse project that failed, it appeared that the model was incomplete. “Our implementation was not carried out through widespread modeling, and there are a few reasons for this:

- the diverse team of experts in robotics, computer vision, software, and control were not all familiar with software modeling techniques;
- the operating environment of real-time behaviors required many components to run on a real-time operating system with limited tool support;
- the behavior of many components is best specified using general-purpose techniques, especially the advanced control algorithms used.

Whether a more experienced modeling team could have been successful is not clear. It appears that reuse with model based development poses challenges for embedded systems.

Under Product Line development, three projects report a failure. In one, the analyst found that the reuse structure was too complex to encourage reuse of software. “Not modifying nonreuse processes, and insufficiently publicizing the repository and the reuse initiative, were the immediate causes of failure.” In the other two, the analyst found that the management was not committed to reuse and that a change in hardware made the software reuse unsuccessful. “As a result, reusable assets were produced, but could never be reused because of changes to requirements, both in functionality and hardware [95] pp. 349-351.”

Reuse in embedded systems was successful for both component-based and unspecified development approaches. Overall, 19 successful projects compare to 4 failures and 6 mixed results.

Nonembedded Systems Table 3.11 also shows the reuse outcomes for non-embedded systems. Outcomes related to *reuse level* are overwhelmingly positive: 35 projects report success vs 10 negative and three mixed outcomes. Lack of success was attributed to a loose development approach, only reusing design and code, no domain analysis, no configuration

Table 3.11: Frequencies of Outcomes

	Product Line		Model Based		Component Based		Unspecified		Total	
Outcome	E	N	E	N	E	N	E	N	E	N
Reuse Level										
1	7	15	10	13	1	6	0	1	18	35
-1	4	2	0	4	0	0	1	4	5	10
0	0	0	3	3	0	0	1	0	4	3
Effort										
1	7	5	3	10	1	4	2	9	13	28
-1	0	0	7	2	1	0	0	0	8	2
0	0	0	6	7	1	2	1	0	8	9
Quality										
1	4	2	2	4	1	6	0	9	7	21
-1	0	0	5	1	0	1	3	0	8	2
0	0	0	6	7	1	2	1	0	8	9
General										
1	13	3	2	8	2	4	2	10	19	25
-1	3	0	1	1	0	0	0	0	4	1
0	0	0	6	6	0	0	0	0	6	6

* Number of projects reporting the outcome
** Note that not all projects reported all outcomes

management, no top management commitment, lack of reuse processes and key roles. In the successful projects, reuse processes, modifying nonreuse processes, management commitment and human factors were all "important for a successful reuse program ([95, p.347])."

Similarly, *effort reduction* was judged positive by 28 projects. Only two reported a negative outcome and nine reported mixed outcomes. Two negative outcomes were for model based development, as were seven of the mixed outcomes. Two of the mixed outcomes were in component based development. No reasons were given for the negative or mixed outcomes.

Quality also improved for most nonembedded systems with 21 reporting positive results, nine mixed results and only 2 negative results. However, as for embedded systems, using a model based development approach had more negative and mixed results ($1 + 7 = 8$) than positive ones (4).

Nonembedded system projects that report *overall reuse success* dominate with positive outcomes (25) vs one with negative and six with mixed outcomes. There also were three positive results using an ontology based approach. Table 3.11 does not list outcomes of projects using an ontology development approach, since embedded systems do not use this approach and hence a comparison is not possible.

The results for nonembedded systems showed predominantly improvement in all outcomes areas, except quality in model based reuse, where the majority of outcomes is mixed or negative.

It appears that positive outcomes are more frequent for nonembedded systems, but negative outcomes for all types of outcome variables still occur for model based development. Note also that product line development is not immune to negative reuse outcomes, specifically, two projects report negative outcomes for reuse level.

The data in Table 3.11 indicate that reuse in nonembedded systems is more likely to be successful for some outcomes, like effort and quality. Overall outcome variables tend to be more positive for nonembedded systems. Particularly there is a smaller proportion of negative outcomes for model based development for reuse level and effort. This is also true for product line development for all outcome variables. Table 3.11 shows that reuse success differs some between embedded systems, not only overall (last columns) but also with regard to successful development approach.

Table 3.12 shows reuse outcomes as percentages for product line (PL), Model Based(MB), Component Based (CB) and unspecified development approaches (UA). For product line development, the reuse level improved in 88% of the projects in nonembedded systems, but only for 64% in embedded systems. 36% of the embedded systems reported lower reuse levels as opposed to only 12% of the nonembedded systems. For both types of systems, 100% of the projects reported improvements in effort and quality. However, in the general impression of reuse experience, 100% of the nonembedded systems projects reported improvement, but only 81% of the embedded systems projects reported improvement and 19% reported negative results. While the reasons for these results were not reported, we

posit that platform dependence and the performance requirements of the embedded systems may have precluded some reuse, and that the reuse level and the general impressions may be related.

Table 3.12: Normalized of Frequencies Outcomes

	PL		MB		CB		UA		Total	
Outcome	E	N	E	N	E	N	E	N	E	N
Reuse Level										
Total	11	17	13	20	1	6	2	5	27	48
+	64%	88%	77%	65%	100%	100%	0%	20%	67%	73%
-	36%	12%	0%	20%	0%	0%	50%	80%	19%	21%
M	0%	0%	23%	15%	0%	0%	50%	0%	15%	6%
Effort										
Total	7	5	16	19	3	6	3	9	29	39
1	100%	100%	19%	53%	33%	67%	67%	100%	45%	72%
-1	0%	0%	44%	11%	33%	0%	0%	0%	28%	5%
0	0%	0%	38%	37%	33%	33%	33%	0%	28%	23%
Quality										
Total	4	2	13	12	2	9	4	9	23	32
1	100%	100%	15%	33%	50%	67%	0%	100%	30%	66%
-1	0%	0%	38%	8%	0%	11%	75%	0%	35%	6%
0	0%	0%	46%	58%	50%	22%	25%	0%	35%	28%
General										
Total	16	3	9	15	2	4	2	10	29	32
1	81%	100%	22%	53%	100%	100%	100%	100%	66%	78%
-1	19%	0%	11%	7%	0%	0%	0%	0%	14%	3%
0	0%	0%	67%	40%	0%	0%	0%	0%	21%	19%

* Per cent of projects reporting the outcome

In model based development, we see that 77% of the embedded systems projects reported improved outcomes. There were no reports of negative outcomes and 23% of the outcomes were mixed. In nonembedded systems, 65% reported improved outcomes, 15% reported mixed outcomes, and 20% reported negative outcomes. However, we find that in effort and quality, the embedded systems projects reported far worse results than nonembedded systems (effort:19% vs 53% positive, 44% vs 11% negative, 38% vs 37% mixed; quality: 15% vs 33% positive, 38% vs 8% negative and 46% vs 58% mixed). The general impression of reuse using model based development was also different, with only 22% positive results for embedded systems compared to 53% positive outcomes for nonembedded

systems. 11% negative outcomes compare to 7% for nonembedded systems; 67% mixed responses for embedded systems compared to 40% mixed responses for nonembedded systems. These appear to be important differences in reuse outcomes for model based development in embedded vs nonembedded systems.

The outcomes for component based development also differed for effort and quality, but the number of observations was too small to draw conclusions.

The overall outcomes regardless of development method is also telling. Overall, the percentage of projects experiencing an increase in reuse level was similar between embedded and nonembedded systems. However, the savings in effort were quite different. While the positive effort outcomes were nearly the same (20% to 21%), the difference in negative effort outcomes was sizeable (40% vs. 14%) as well as in mixed outcomes (40% vs 64%). The difference in quality outcomes was also striking, with nonembedded systems reporting positive outcomes twice as often as embedded systems (66% vs. 30%) and embedded systems reporting negative outcomes nearly six times as much as nonembedded systems (35% vs 6%). General reuse experience was also noticeably different between embedded systems and nonembedded systems, with embedded systems reporting a positive outcome for 66% of the projects and nonembedded systems reporting a positive outcome for 78% of the projects, while embedded systems reported negative outcomes in 14% of the projects. Nonembedded systems reported negative outcomes for only 3% of the projects. The reasons for this cumulative difference are not clear. However, one study reported one failure in quality differences between the ground system (nonembedded) and the flight system (embedded) that was related to the size, reliability and complexity of the software. While there was no discussion about which, if any, of these contributed to the failure, the conclusion was, "We found that development strategies in general performed as well or better than drastic change strategies on ground software, but did worse than adopting no strategy in the case of flight software systems ([101])."

Testing for significance. Our next question was whether the differences we observed and analyzed above are strong enough to be statistically significant. We performed a Chi-Square test on the outcome scores. Table 3.13 shows the results. This indicates that the differences in reuse success described above are strong enough to be statistically significant for two situations:

1. savings in effort are less likely to be successful for embedded systems when a model based approach to reuse is employed
2. quality improvements are less likely to be realized in embedded systems (in studies that did not report on the development approach used).

Unfortunately, the papers do not provide a solid chain of evidence to identify reasons.

Table 3.13: Chi Squared P-Values Embedded Systems vs Nonembedded Systems

Outcome	**Product Line**	**Model Based**	**Component Based**	**Unspecified Approach**
Reuse Level	0.1213	.2291	N/A	N/A
Effort	0.2609	**0.0292**	.3779	0.0704
Quality	N/A	0.2138	.2062	**0.0005**
General	0.3059	0.2583	N/A	N/A

Since we did not have performance scores for nonembedded systems, we had to eliminate this success factor from the analysis. For some combinations of criteria and development strategies, the sample size was too small, hence the corresponding table entries are marked N/A.

3.6 Threats to Validity

Since much of the analysis is better classified as qualitative, we assess the following types of validity: descriptive validity, interpretive validity, theoretical validity, generalizability, and evaluative validity [90].

Descriptive Validity Descriptive validity relates to the quality of what the researcher reports having seen, heard or observed. Because observations are important, we needed to include grey literature. Since these are not always peer reviewed, the rigor of data collection or reporting is uncertain. To alleviate this threat to validity, we used SEI sources. While not peer reviewed, it is highly respected in industry. However, even in peer reviewed material, the reporting might be subject to mis-interpretation of what was observed.

We investigated how the researchers handled information, including their presentation of data, method, hypothesis, analysis, threats to validity and significance. Of the 84 initial papers, half of the studies did not include the data that was claimed to be analyzed. Only five had a hypothesis. Only nine discussed threats to validity. Many studies are reports on reuse efforts undertaken by industry in specific approaches or products they were marketing. Many of these studies contained few or no metrics and they did not use hypothesis tests, analysis of their results or clues about validity issues that might have affected outcomes. There was a great deal of expert opinion not backed up by metrics. Even in the very complete discussions of developers' attitudes to reuse, there was no discussion as to whether the way the developers felt about their reuse experience was backed up by quantitative results.

An important weakness is the failure of many studies to include enough metrics of outcomes, leading, by necessity, to few projects that could be reported on in section 6. Another weakness is the failure to include performance metrics in studies of nonembedded systems. As a result of this, some of the key differences may not be discovered. While the absence of stringent performance requirements may not affect reuse success in nonembedded systems, it makes it difficult to study embedded systems vs nonembedded systems in terms of reuse.

Interpretive Validity The second threat to validity is concerned with what objects, events, and observations mean to the experimenters.

The first challenge in conducting this research was understanding development strategies and system types as variables. It was important to identify where system types and development approaches overlapped. For instance, a system as an end product could be embedded, but the simulation software and many subsystems could be nonembedded software. This required the ability to determine if the empirical study focused on the embedded portion of the system or on a nonembedded portion. In addition, the development strategies overlapped. Where more than one development approach appeared to exist, a determination had to be made as to whether the development approach was truly a combination or if one or another development approach dominated or even if one development approach subsumed others.

We found no studies whose major focus was comparing reuse in embedded systems vs nonembedded systems. Even when reuse in both types of systems is studied, the authors aggregate the results. This may have skewed the results reported in these studies. To alleviate this threat, we analyze each project individually.

Another interpretive threat to validity is the lack of common, consistent definitions. This is especially true with model-based development strategies. While there are many types of models, the empirical studies are not clear what types of models are being used, whether they are architectural models, design models or behavioral/performance models. It is also unclear, in empirical studies of component based reuse, whether the term component means the software with the platform or the software alone. In the grey literature mentioned above, the impartiality of the reporting is also uncertain. The mitigation, again, was to use respected industry sources, even though they are not peer reviewed.

In order to analyze the data, it was important to find a way to convert opinions into analyzable data. This was because quantitative data was scarce. It was also important to determine reuse outcome categories, since every paper had its own perspective and thus identified its own metrics (insofar as they used metrics). Categorizing metrics based on similarity required careful reading of each paper. The different viewpoints of different studies may have affected data interpretation (lack of clarity for our purposes).

Due to lack of information about system types, we had to exclude a series of studies. Had we known the system type in the studies, we could have had more data.

Finally, there was the question of the validity of subjective opinion as data. In many experience reports and expert opinions, the objectivity of the author may come into question. A few were possibly marketing. Setting the threshold for inclusion at 20% empirical content may have led us to include too many empirical papers that lack rigor, but setting it higher would have further reduced available data.

Theoretical Validity Theoretical validity refers to an account's validity as a theory of some phenomenon. It depends on the validity of the construct of the experiment and on the validity of the interpretation or explanation of the observations. It also depends on whether there is consensus within the community about terms used to describe context, events and outcomes. With the number of similar but not identical metrics presented, and ways of measuring success, there is a threat to the validity concerning the similarity or difference of the perceived value of reuse. This was mitigated by using a common scoring system, resulting in ordinal rather than ratio metrics.

Generalizability "Generalizability refers to the extent to which one can extend the account of a particular situation or population to other persons, times, or settings than those directly studied [90]." It is comparable to external validity used in quantitative studies. In this situation, the threat to validity consists of the access to research that has been performed. Corporations may be performing studies that they choose not to release to the public. This could be because of the proprietary nature of the information, or because publication is not a corporate focus. The information in those studies could reveal factors not uncovered in the published material. There is a tendency to publish successes and to keep working on or terminate efforts that have failed, leading to overreporting of success. This also leads to the low numbers of projects available for comparison.

In some cases, the reuse process was being applied for the first time. This could have been one reason there was so much evidence of success through reuse. In these cases, it is

fair to ask how much of the improvement is because there is SOME process being followed as opposed to none at all (in other words, was reuse the reason for improvement or was it the existence of a process?)

Evaluative Validity Evaluative validity refers to the evaluative framework in deciding whether the reuse was, in fact, successful or not, and if so how much. The frameworks were likely to have differed in the different studies because the contexts were different. This was mitigated by considering both the researchers' evaluation of reuse and the observations upon which the evaluations are based.

3.7 Conclusion and Future Work

We analyzed empirical studies of reuse dating from the time of DoD's release of the DoD Software Reuse Initiative (1992). We considered five development approaches to reusing software in both embedded and nonembedded types of software systems: ontology, product line, model based, component based and unspecified approach (where the development approach was unknown). We considered eight different study types. Out of 84 candidate papers, only 43 had enough usable empirical content to enable a comparison of reuse outcomes in embedded vs. nonembedded systems.

Reported studies (experiments, case studies, surveys, etc) from industry as well as academia are surprisingly few. While we found a wealth of papers describing reuse, we found few with hard evidence either to the benefits or lack of benefits from reuse addressing or distinguishing between reuse in embedded vs nonembedded systems. We also found a number of papers questioning whether the benefits exist.

We divided the reuse studies into categories of embedded, nonembedded, and both embedded and nonembedded systems. We grouped these into subcategories based on development approach. Finally, we grouped them by type of empirical study.

Having catalogued the empirical studies, we proceeded to analyze individual projects in these studies. This allowed a first set of comparative analyses. We analyzed what reuse outcomes were reported and how they were measured. We mapped reported outcomes into a three point Likert scale as success, failure or mixed result for reuse outcomes related to amount of reuse, effort, quality, performance (where available) and general reuse success. We compared embedded and nonembedded systems with regard to the proportion of positive, negative and mixed reuse outcomes for each of the different development approaches. We also performed an overall comparison between reuse in embedded and nonembedded systems.

We found that results from reuse studies of nonembedded systems were not necessarily extendable to embedded systems. For example, in a model based development approach, effort outcomes were significantly more positive for nonembedded projects than for embedded projects. In order to suggest that a development approach or method is generally effective, it needs to be studied in both types of software systems. This research casts doubt on whether it is wise to extend findings from reuse studies of nonembedded systems to embedded systems.

Limitations of existing studies point to a lack of solid metrics to underpin the many opinions about reuse. While we were able to translate qualitative responses to ordinal data for comparative analysis, the power of nonparametric tests is lower than for parametric ones and may have prevented us from finding more differences that are statistically significant. It would be helpful to have research with reuse outcome measures that allow statistical tests with a higher power against which to test the hypotheses.

In embedded systems, where the source code is optimized against a processor, the reuse of source code may not generate much savings, because when the processor is changed, the code must be reoptimized. We saw symptoms of this in reports of unsuccessful reuse levels (i.e. less reuse than expected) in some of our data.This could contribute to the lack of success of reuse as reported in the reuse outcomes for model-based development.

Another weakness in existing studies is the failure to include performance metrics. While the absence of stringent performance requirements may not affect success in nonembedded systems, it is difficult to compare embedded systems with nonembedded systems in terms of reuse when performance requirements must be met.

Based on our analysis, we have the following recommendations:

- If a company is interested in reusing embedded software, the existing empirical evidence suggests that it would be prudent to proceed carefully, and treat initial reuse attempts as pilot studies, rather than presuming that reuse benefits are a foregone conclusion.
- The tight connection between hardware and software in embedded systems, taken with the lack of performance metrics in existing reuse studies relating to embedded systems, suggests that embedded software reuse is less risky when the hardware is identical, or has characteristics which will yield improved performance when hardware and software are taken together (for instance, faster processor speeds with fully instruction-compatible processors).

An area for potential research would be to study whether an ontology-based approach would be of value in embedded systems. More research on combinations of approaches is also useful. Yet another potential area of research could be which types of models work best for model-based development strategies, and whether they are the same for embedded and nonembedded systems.

Chapter 4

Survey

Having analyzed the empirical literature, we found no work that compared embedded to nonembedded systems. We decided to do a survey of practitioners to see if they could shed some light on these questions.

Software reuse has been the subject of empirical studies for decades. There have been studies of reuse methods, success factors, approaches and tools. However, while modern reuse strategies have been studied, these reuse strategies have not been compared with each other. Similarly, reuse in embedded and nonembedded systems have been studied, but they not compared to each other. This chapter describes the results from a survey taken at a major aerospace corporation, in order to characterize developers' views on software reuse. A total of 78 developers participated in the survey. The questions focused on modern reuse strategies and their effectiveness, whether they were used alone or in combination. It also analyzes reuse when developing embedded systems vs. nonembedded systems.

4.1 Related Work

An extensive survey of literature dating back to 1992 [7] found 84 empirical studies focusing on software reuse including five surveys (based on questionnaires or structured interviews). The important findings of the surveys were:

Li et al. [86] investigate the state of practice and industry-wide trends in reuse of in-house built components with respect to requirements (re)negotiation, component repository, component understanding and component quality. They also study the relationship between the companies' reuse level and these factors. A 12 question survey was given to 30 software developers in three companies. 17 developers responded. There were six research questions: Does requirements (re)negotiation for in-house components really work as efficiently as people assume? Does the efficiency of component related requirements (re)negotiation increase with more in-house built components available? Does the value of component repository increase with more reusable components available? How can a component user acquire sufficient information about relevant components? Does the difficulty of component documentation and component knowledge management increase with increasing reuse level? Do developers trust the quality specification of their in-house built components? If the answer is no, how can they solve this problem? Each research question was addressed by one or more survey questions. After a summary the answers related to of each research question (three via tabulation, one using bar charts, two summarized the information). Ordinal values were assigned to the responses, and a Spearman Rank Correlation Coefficient analysis was used to determine significance. Renegotiation of requirements was important, but will probably not increase efficiency with increased reuse level. A component repository was not a key factor in successful reuse. Most developers were not happy with existing documentation and encouraged more attention be placed on informal channels for information about components being reused. This was a study of component based reuse in nonembedded systems. No qualitative data were collected.

Rothenberger et al. [111] explores practices for code reuse. Their goal is to identify practices in reuse that can be proactively used in formulating a well-thought-out reuse strategy. The study uses survey data from software development groups working with software reuse to reduce numerous reuse success factors identified in earlier studies to a set of six reuse dimensions: planning and improvement, formalized process, management support, project similarity, and object technologies. The survey consisted of 20 questions linked to 8 reuse

practice categories: Project similarity, reuse planning, measurement, process improvement, formalized process, management support, education, object technologies, common architecture and commonality of architecture. 71 survey responses from 67 projects were used in the analysis. The responses were analyzed by grouping co-occurring practices into single constructs using Principle Component Analysis (PCA). PCA reduced the six dimensions to five. Object technologies did not explain enough variance to be included. System types and development strategies were not discussed. There were no qualitative questions to explain causes for the observed behavior.

Morisio et al. [95] identified some of the key success factors in adopting or running a company-wide software reuse program. 24 projects between 1994 and 1997 were analyzed using structured interviews from 32 Process Improvement Experiments, usually with the project manager. The answers were tabulated for observation. They looked for a correlation between independent variables (state variables, like staff experience, size of baseline, domain, etc. and high- level control variables, like human factors, mamagement commitment, etc.) and the dependent variable (success or failure). The interviews consisted mostly of closed questions. Data were coded and organized into categories of success and failure and then analyzed using correlation tree analysis. Top management commitment was "a prerequisite for successfully designing and enacting process change." Of the state variables, only "type of software production" had impact. Three main causes of failure were not introducing reuse-specific processes, not modifying nonreuse processes, and not considering human factors. Based on these results, Morisio et al. suggest a process of how to introduce reuse in an organization. While both embedded and nonembedded systems were identified in this paper, the results were analyzed together, but they were not compared against each other.

Frakes and Fox [41] used results of a survey to predict the likelihood of certain reuse levels of later life cycle objects from earlier ones, and reuse levels of life cycle objects given reuse levels of other life cycle objects, whether from a preceding phase or not. 113 software practioners from 29 projects answered questions about their own and their organi-

zations' reuse practices and policies. The companies' primary businesses included software, aerospace, manufacturing, telecommunications, as well as universities. Respondents rated reuse levels of requirements, design, code, test plans, test cases and documentation. Reuse levels were measured on a 10 point Likert scale for each artifact. The data were analyzed using Pearson's correlation coefficient for reuse levels of different artifacts. The data shows strong, significant positive correlations between reuse levels of all life cycle objects. From the Pearson's Correlation Coefficients, Frakes et al. concluded that reuse of artifacts early in the life cycle was predictive of reuse of artifacts later in the life cycle. There was no discussion of whether the projects used embedded or nonembedded systems, nor were the development approaches discussed. No qualitative data were collected to explain causes for observed behavior.

Rine and Sonnemann [108] developed a nine question survey to identify which factors are predictive of reuse success. This survey was used to measure software reuse capability, productivity, quality and the set of software reuse success factors (management approach, software architecture, availability of components, and quality of components). The questionnaire was conducted as a mail survey. It generated 109 responses from 99 projects in 83 organizations. Software reuse was measured as a percentage, or frequency, of the components reused. The researchers chose to use F-tests to further analyze the data. From the survey results, researchers identified eleven factors predictive of reuse success. A product line development approach, common architecture, including data formats were important to success. The data also showed a strong relationship between software reuse success and productivity. Based on these results, they suggested ways to invest in reuse capabilities.

While one survey did include both embedded and nonembedded systems, none compared reuse success between them. Similarly, while some studies considered development approaches, they did not compare reuse success for them.

4.2 The Survey

4.2.1 Context, Research Questions, and Hypotheses

In our research, we have studied the history of reuse in the Aerospace industry in terms of how reuse came into being, how it has evolved and trends in how reuse is now used. We reviewed existing empirical studies about reuse and compared the reuse outcomes for embedded systems against nonembedded systems using different development approaches. We discovered in the review of existing literature that reuse in embedded systems leads to significantly less positive outcomes than in nonembedded systems when the development approach is model based engineering, and that, overall, reuse in embedded systems is less successful than reuse in nonembedded systems.

There were also some indications that some of the difference in outcomes could be related to the artifacts reused, but few of the empirical studies focused on artifacts and their impact on reuse. In fact, while not explicitly stated, it appeared that most of the projects studied in the literature were reusing code. When other artifacts were considered, their impact on the success of the reuse was not studied.

Our search of existing literature left several questions that we felt needed to be answered in order to understand and implement successful reuse. First was whether industry practitioners share the same reuse experience as the research for embedded and nonembedded systems. Second is the question of what artifacts can be reused for successful outcomes. Third is the whether there is a difference between embedded systems and nonembedded systems in outcomes. We also wondered whether the size of the project influences reuse outcomes and whether the developer expectations influence the outcomes in reuse. Thus, we established our research questions:

RQ-1 Do embedded systems use different development approaches than nonembedded systems? This question tries to identify what development approaches are being used on embedded systems vs nonembedded systems projects and how effective they are with regard to reuse. We also want to discover whether reuse strategies are used in combination.

Reuse in various development approaches have been studied singly, but are there additional benefits of using the development methods in combination? The null hypothesis assumes that the same development approaches are used whether the system is embedded or nonembedded.

RQ-2 Do embedded systems reuse different artifacts than nonembedded systems? What artifacts are being reused on what types of projects? Are there artifacts more commonly reused in embedded systems than in nonembedded systems? The null hypothesis assumes that embedded and nonembedded systems reuse the same artifacts at the same level with comparable outcomes.

RQ-3 Do reuse outcomes vary between embedded systems and nonembedded systems? This question is central to our research. It is designed to determine whether there is a difference in reuse effectiveness or preferred development approaches based on project type. Positive outcomes are measured by: fewer labor hours, fewer defects, less test time, fewer items to be tested, and less risk. While many aspects of software reuse have been studied deeply, our literature review did not surface this specific comparison. The null hypothesis assumes that there is no significant difference in the effectiveness of reuse whether the project is an embedded or a nonembedded system.

RQ-4 Does reuse effectiveness vary with project size? This question tries to identify whether the size of a project makes a difference in the reuse strategy best suited to it. It also tries to determine whether there is a point at either end of the size range where reuse does not confer benefits such as higher efficiency and quality. Our null hypothesis is that reuse is equally effective across the size spectrum.

According to [131], the survey is the research method best suited for answering questions whose form is who, what, where, how many and how much. It focuses on contemporary events and does not require control of behavioral events. Surveys are "are also well suited to gathering demographic data that describe the composition of the sample. Surveys are inclusive in the types and number of variables that can be studied, require minimal investment to develop and administer, and are relatively easy for making generalizations.

Table 4.1: Survey Rationale

Purpose	Rationale	Questions
Respondent Information	RQ-1 The purpose of these questions is to correlate reuse experience with the type of engineer (i.e. hardware, software, systems), the company (which corresponds to the types of programs and the culture), and the experience level of the engineer.	SQ1-4
Project/ Application Information	RQ-2, RQ-3 The purpose of these questions is to correlate the size of the program, the nature of the system/program, the software type. This should offer insight into whether embedded and non-embedded use the same strategies and whether successful strategies are similar.	SQ 5-9
Reuse Information	RQ-4, This set of questions helps identify the type of reuse strategy employed, whether success is improved with being part of the decision, and which products are reused (details of the strategy) and whether the program is far enough along to measure factors that occur late in the program. We are able to compare development strategies and artifacts used on embedded systems vs. nonembedded systems.	SQ 10-14
Reuse Effectiveness Information	RQ-4 This set of questions will help correlate the effectiveness of the strategy against the strategy by identifying and scoring the change in outcomes attributed to reuse.	SQ 15-20
Reuse Experience	RQ-5 This set of questions will help analyze the user the experience of the reuse approach	SQ 21-22

Surveys can also elicit information about attitudes that are otherwise difficult to measure using observational techniques [52]."

4.2.2 Procedure

Our first step was to identify the type of empirical study to answer the research questions. We determined that a survey would be best suited for our purposes. Surveys "are advantageous when the research goal is to describe the incidence or prevalence of a phenomenon or when it is said to be predictive about certain outcomes [131]." The form of the questions best suited for surveys matched our own questions, we are looking for current reuse experiences, and we want to know what is happening in practice without inserting controls.

Table 4.2: Survey Plan

Step	Our Approach
Rationale	Determine whether reuse strategies and approaches were the same for embedded and nonembedded systems
Focus of Study	Aerospace Engineering Practitioners, System Type, Development Methods Used, Artifacts Used, Outcomes
Sampling Plan	Subject Selection - Members of Listserves Medium - Survey Hosted on Corporate Website Variables to Measure - Development Approach, Artifacts, Outcomes Resource Limitations - Access to subjects
Instrument Development	Survey Questions Introduction to Questionnaire Evaluation, Review by Experts, Pilot
Administration	Web based questionnaire One Month for answers Tabulate Answers
Data Validation	Review responses for reliability Group as required
Analysis Plan	Frequency Tables, Box Plots, MANOVA, PCA, Free-form response analysis
Reporting Plan	Dissertation, publication

* Survey plan elements derived from [52], [112] and [129]

Responses to quantitative questions in the survey can be analyzed with statistical analysis techniques. Answers to qualitative questions are sources for potential explanations for statistical results. The qualitative responses in our survey are used to obtain potential explanations for the quantitative answers. Glasgow [52] points out that "survey instrument development must be preceded by certain prerequisites. First, the focus of the study must be carefully defined. Second, the study objectives must be translated into measurable factors that contribute to that focus. Finally, the survey must be consistently administered."

Based on the research questions discussed above we developed a survey to collect both qualitative and quantitative information, leading to a mixed method survey. It was developed in accordance with [52], [112] and [129], shown in Table 4.2.

4.2.3 Focus of Study

In order to answer the questions, we decided to focus our study on aerospace and defense engineering practitioners. We wanted to learn about the differences and similarities in reuse

outcomes for embedded vs. nonembedded systems. In particular, we wanted to know about the use and impacts of development methods they are currently using in their reuse efforts and of artifacts they were reusing.

We selected a corporation in the aerospace and defense industry that was composed of multiple companies dispersed across the United States, with a variety of technologies and projects for a number of different government agencies. The projects in this corporation offer a cross section of large development efforts in the industry and research to support these large efforts.

4.2.4 Sampling Plan

Having decided on a survey as our next study, we needed to determine who we should survey and how the survey should be conducted. We developed a sampling plan to describe the approach used to select the sample and the choice of media through which the survey will be administered. The sampling plan is the methodology that will be used to select the sample from the population [52].

This sampling plan identified the desired sample of engineers based on the variables we wanted to measure (system type, development approach, artifacts used and outcomes), the estimates required, the reliability and validity needed to ensure the usefulness of the estimates, and our resource limitations related to the conduct of the survey.

A major consideration in the sampling plan involved access to sufficient subjects working a variety of project types and the ability to generate enough samples to aggregate responses for meaningful analysis. Other considerations included the ease of administering the survey for the researcher, convenience of taking the survey for the subjects, anonymity and comfort. These considerations led us to select a written questionnaire, as the appropriate method. This questionnaire could be delivered via an easily accessible web site using a web-based survey tool. It is hosted online at a site used by the corporation to share information. All engineers in the company can access this website. Once the survey was approved and released, emails were sent to systems and software engineers via listserves

requesting their participation. Since the number of participants on these listserves is unknown and may have some overlap, the total number of requests and hence the response rate is unknown. This survey uses non-probability sampling, based on convenience and self selection.

While the disadvantages of written surveys include their subjectivity to certain types of error such as coverage error (where the response rate is not balanced or the questionnaire does not reach certain areas), nonresponse error (where some candidate subjects do not participate), bias and item nonresponse (where some questions may be inadvertently or intentionally skipped), we felt the advantages outweighed these types of errors.

4.2.5 Instrument Development

We identified the types of information needed, shown in the first column of Table 4.1. These types of information are Respondent Information, Program/ Application Information, Reuse Information, and Reuse Effectiveness Information. The second column of the table explains the reason for seeking the information. From this, we developed survey questions, tailoring each survey question to study one of the research questions.

The questionnaire (Appendix A) consists of a combination of check boxes and short answers, giving subjects the opportunity to contribute their own qualitative input as well as easily measured responses. Based on the research questions discussed above we developed a survey that could collect both qualitative and quantitative information. The survey was developed in accordance with [88] and [129]. The qualitative responses in the survey are used to identify correlations across the data. This research method allows the opportunity to obtain satisfactory amount of information from each respondent with the help of a structured survey. The quantitative questions in the survey give us the ability to analyze data with statistical tools and analysis techniques.

Each respondent was asked about their most recent reuse project. The questionnaire started with collecting information about the respondents, the project they were reporting on, how their project was employing reuse, their perceptions of the effectiveness of that

reuse and their overall experience with reuse. An initial set of survey questions was developed to translate those objectives into measurable factors. As recommended in [52], careful attention was paid to the structure of the questions:

- Wording - The wording of the questions had to be clear, understandable to the respondents. Wherever there could be ambiguity, the question explained what the researcher meant in the question. For example, in question 4, the question "What type of program are you working on?" was elaborated with "(i.e. what is the final product?)." In addition, the questions should be asked in a way that does not lead the subject or his responses. The questions should not include a predisposition for or against any particular perspective. This avoids the threat to validity called "biased wording."

- Feasibility - The questions should not seek information the mid level engineer would not have the ability to answer, such as management data.

- Ethical - The questions should not involve disclosure of proprietary information or personal information or information the engineer may feel uncomfortable discussing.

Questions about the demographics of the respondent (for example, type of engineer, type of project, years of experience) and seeking information about the projects (for example, system type, development approach, artifacts used) were answered on a nominal scale. Some of these questions would only accept a single answer (years of experience), while others allowed the respondent to select as many answers as applied (artifacts used). Questions about outcomes (for example, improvement in labor hours used, reduction of defects) were answered on an interval scale These quantitative questions in the survey give us the ability to analyze data with statistical tools and analysis techniques.

There were also some open ended questions in the form of questions asking for free form answers (text fields) that could provide insight into the reasons for the responses. This gives the subjects the opportunity to contribute their own qualitative input in addition to the easily measured responses. Table 4.1 shows the general information we wanted to collect

in column 1, column 2 explains why we were collecting the information, and column 3 identifies the questions in the survey developed to obtain this information. The questions in the survey are tied to the research questions as shown in the table. The qualitative responses to these questions are used to identify causal relationships.

The questionnaire was reviewed by 4 colleagues to obtain comments and to ensure that the questions were understandable and would obtain the desired information. These colleagues are fellows and distinguished engineers from different divisions and locations throughout the corporation. They suggested some modifications, which were implemented. This enhanced accomplishing the third Mitre requirement of ensuring the questions were developed by those versed in the topic. The survey was entered into the survey tool on the web site and a few individuals piloted the survey, as recommended by [52]. From the experiences of the pilot subjects, a few changes were made to make the survey more understandable.

Next, an introduction was created explaining the purpose of the survey and its proposed uses. Ethical considerations were also addressed: The introduction stated that the survey was anonymous and voluntary, and that responses could be deleted at any time. Finally, the introduction defined terms used in the questionnaire.

4.2.6 Administration

The medium for delivery that we selected consisted of a web site used by the corporation to share information across companies, divisions and locations, with an embedded survey tool. All engineers in the company have permission to access this website. This web site was accessible by all engineers in the corporation. The sampling approach would be self selection, in that all engineers would be informed of the existence of the questionnaire and any engineer who chose to participate could do so. This conforms to the Mitre requirement that the survey be consistently administered.

Once the questionnaire and introduction were approved and released, emails were sent to systems and software engineers requesting their participation. This survey is a non-

probability sampling, based on convenience and self-selection. The engineers that participated in the survey were employees in various companies and locations, reusing assets developed by the various sites for various projects. The questionnaire was available for a month. Responses were collected in an Excel spreadsheet, one line per respondent.

4.2.7 Data Validation

Once the responses were downloaded in the spreadsheet, they were reviewed. We counted the respondents from embedded systems and from nonembedded systems, as well as across domains and areas of expertise, to ensure we had enough in each catagory for comparison. In cases where the answers were very specific but could be generalized to a higher grouping, they were generalized. For example, when asked what domain they worked in, several respondents replied with very specific specialties within domains (i.e. orbital analysis). These answers were then allocated to a higher level domain (Guidance, Navigation and Control). Because of the limitations of the tool, some answers were written in a text field rather than selecting from a predefined answer. These were converted to the appropriate predefined answer. In one case, the question "How large is your program in KLOC" it was apparent that some had answered in LOC. Since we could neither be sure at this point which were correctly in KLOC and could not return to the respondents, we had to throw this question out due to questionable reliability.

4.2.8 Analysis Plan

We first perform descriptive analysis, using spreadsheets, bar charts, pie charts and box plots. The frequencies of answers were normalized to percentages for direct comparison.

In preparation for quantitative analysis, the questions were coded. For yes/no questions, the coding recorded a 1 when the item was selected and a 0 when it was not. For example, for development approach used, if the respondent indicated product line, component based development and ad hoc reuse were all used, product line, component based development and ad hoc would each receive a 1, where model based and ontology based development

would receive a 0. The questions with Likert scale measures used a scale of 1-6.

Next, free-form answers were studied to provide reasons for the quantitative findings. Comments were studied for common words, phrases, and concepts and grouped into categories that might offer insight into those findings.

Finally, we performed a Principle Component Analysis. The reason for using PCA was to identify if there were any development approaches or reuse artifacts that tended to be used together.

4.3 Results

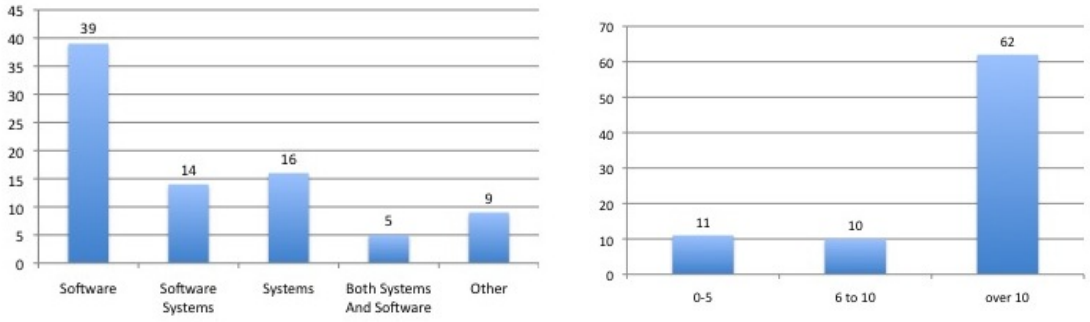
First, answers to quantitative questions are analyzed via descriptive statistics. Then we test our null hypotheses (i.e. there is no difference in reuse practices and success between embedded and nonembedded systems) using MANOVAs. A Principal Components Analysis was performed to analyze for commonalities and differences in attributes. Qualitative information provided by our respondents helps explain some of our results and provides reasons why reuse was or was not successful.

4.3.1 Descriptive Statistics

78 engineers responded to the survey. Demographic questions included: What is the experience level of the survey respondents? What type of engineer is responding? What type of system is it? The first question is whether experience in the industry affects how an engineer looks at reuse. This includes years of experience, the kind of work, and the type of system.

The respondents' experience ranged from less than a year to over 30 years. Over 75 % of the respondents had more than ten years of experience (62). 10 had 6-10 years of experience and 11 had five years or less. The respondents were software engineers(39), systems engineers(11), software systems engineers(14), both systems and software engineers(5), planners, managers and business development engineers (9). We thus had a majority of

Figure 4.1: Demographic Information



(a) Respondents by Discipline

(b) Years Of Experience

very experienced professionals, most of whom had technical expertise.

Table 4.3: Systems and Applications

System Type	No	**Application**	No
Satellites	22	C3, Comm, GNC	18
Aircraft/Helicopter/Avionics	17	Data, Information	9
Missiles, Rockets	10	Simulation, Test	9
Ground Station/Support	9	Mission	8
Data Collection	5	Algorithm Devt	7
Non Satellite Space	4	Hardware	7
Other	12	Web-Based	7
		Other	18

The subjects reported on embedded and nonembedded systems, ranging from helicopters to logistics. The distribution is shown in Table 4.3. The applications range from algorithm development to domains such as Guidance, Navigation and Control, Communications and Command, to web based applications. The "other" catagory includes application types receiving three or fewer responses, including architecture, graphics and business. This shows a good cross-section of application types with some emphasis on satellites and avionics. The latter heavily emphasizes embedded systems.

RQ-1 Do embedded systems use different development approaches than nonembedded systems? The survey asked about reuse in the development approaches or combinations of those approaches being used. Each respondent reported on one project. The respondents

were given the option of choosing as many of the development approaches as applied. Surprisingly, four projects, three nonembedded software systems and one embedded software system, said they were employing reuse but without a specific approach, not even ad hoc. Maybe their approach was not in the list. We removed these responses from further consideration. Of the 78 subjects, 41 reported on embedded systems, while 37 reported on reuse in nonembedded systems.

Table 4.4: Development Approach by System Type

Development Approach	E	N
Ad Hoc	8	8
Component based	3	0
Component based+Ad Hoc	4	2
Component based+COTS	0	1
Component based+Ad Hoc + COTS	1	0
Component based+Product Line	1	2
Component based+Product Line +COTS +Ad Hoc	3	0
COTS	1	4
COTS+Ad Hoc	2	3
Model based	1	2
Model based+Component based	0	1
Model based+Component based+Ad Hoc+COTS	2	3
Model based+Component based+Product Line	2	0
Model based+Component based+Product Line +Ad Hoc	3	1
Model based+Component based +Product Line +Ad Hoc +COTS	3	1
Model based+Ad Hoc	1	0
Model based+Ad Hoc+COTS	1	0
Model based+Product Line+COTS+Ad Hoc	0	1
None	1	3
Product Line	4	2
Product Line+Ad Hoc	0	1
Product Line+COTS+Ad Hoc	2	3

* E = Embedded, N = Nonembedded
** Ad Hoc=Ad Hoc/Legacy/Heritage, COTS=COTS/GOTS

As shown in Table 4.4, 33 projects used only one development approach, not including the four that used none at all. Of these, 17 were embedded systems, 16 were nonembedded systems. The rest used a combination of approaches. While nonembedded systems were more likely to use ad hoc, COTS/GOTS, and even no approach, embedded systems were more likely to use component based, ad hoc and product line development approaches. This

was also true when those development approaches were included in the combinations of approaches. There were 22 different combinations of approaches, and no one combination approach was used by more than four projects. It is particularly interesting to note that most of the composite approaches were used by only one or two projects. No ontologies were reported.

What is interesting to see is that so many respondents reported employing more than one development approach during reuse (recall, they report on a single project). This is not common when developing systems from scratch. What needs to be considered is that in the aerospace industry, some systems evolve over decades, development approaches change and new functionality and enhancements may follow more modern development approaches. Even with a single project, lack of standardization, diversity of divisions contributing to a project and the large size of some of the projects can also lead to this phenomenon. Further, different subsystems (e.g. guidance, navigation and control vs power control) may find different approaches more useful, e.g. performance models and autogeneration of code vs. COTS. Note also that both embedded and nonembedded systems appear to employ ad hoc reuse to the same degree, this is not the case for other (combinations of) approaches.

Next, Table 4.5 summarizes how often a specific development approach was used (alone or in combination with others). Column 1 lists the development approach. Columns 2 and 3 list the frequency of each development approach for embedded (E) and nonembedded (N) system projects. Columns 4 and 5 report the proportion of projects using a given development approach, either alone or in combination with others.

The most commonly used development approach for embedded systems is component based development, while nonembedded systems included Ad Hoc reuse and a Product Line approach most frequently. Embedded systems projects used component based, model based and product line approaches more frequently than nonembedded systems, while nonembedded systems used Ad Hoc reuse and a COTS/GOTS approach to reuse more than embedded systems. Interestingly, the most commonly used approaches in embedded system projects are not the most commonly used in nonembedded system projects and vice

Table 4.5: Development Approach Contained in Strategy

Development Approach	Frequency		Relative Frequency	
	E	N	E	N
Ad Hoc	11	15	0.27	0.41
Component Based	22	11	0.54	0.30
Model Based	13	9	0.32	0.24
COTS/GOTS	12	15	0.29	0.41
Product Line	18	11	0.44	0.30

* E = Embedded, N = Nonembedded

versa. We will investigate in the next section whether these differences are statistically significant.

RQ-2 Do embedded systems reuse different artifacts than nonembedded systems?

Next we turn to the artifacts used by embedded and nonembedded systems projects. The artifacts we selected to study are requirements (shall statements in specifications), architecture (allocation of capabilities to configuration items), models (including performance models, architecture models, environmental models, simulations, and test models), use cases (sometimes called scenarios), code, drawings (like schematics, blueprints, wiring diagrams), hardware (like processors, sensors, platforms), test products (like test drivers, test data, seeded test data), and test clusters (components, whether software, hardware or hardware and software, that have previously been integrated and tested together).

Table 4.6 shows the number of times these artifacts were reused in embedded and nonembedded system projects (columns 2 (E) and 3 (N)) respectively. Respondents checked as many artifact types as they reused on their project. Since the number of respondents is different for embedded (41) and nonembedded (37) systems, we report the proportion of projects reusing a given artifact (reuse level). The most often reused artifact is code, followed by requirements, and architecture. This is also the most commonly reported combination.

Most strikingly, reuse levels for all artifacts is higher in embedded systems projects than nonembedded ones. Sometimes the difference is small (reuse of models, drawings),

Table 4.6: Reuse Artifacts Contained in Strategy

Reuse Artifacts	Frequency		Relative Frequency	
	E	N	E	N
Requirements	29	19	0.71	0.51
Architecture	29	18	0.71	0.49
Models	17	13	0.41	0.35
Use Cases	16	6	0.39	0.16
Code	36	26	0.88	0.70
Drawings	9	7	0.22	0.19
Hardware	22	10	0.54	0.27
Test Products	21	10	0.51	0.27
Tested Clusters	4	0	0.1	0.0

* E = Embedded, N = Nonembedded

but often reuse levels in embedded systems are at least 10% higher higher than in nonembedded systems (e.g. requirements, architecture, use cases, hardware, test products and test clusters). This may point to a reluctance in embedded systems to embark on large-scale change, since systems are tightly coupled to hardware. It may also point to longeevity of products (i.e. avionics) with only incremental rather than sweeping changes. This does not necessarily mean that reuse is easier, just that it is different. The question is whether these differences are statistically significant. We investigate this in the next section.

RQ-3 Do reuse outcomes vary between embedded systems and nonembedded systems?

Table 4.7 shows the savings or costs respondents reported in labor (columns 3 and 4), defect reduction (columns 5 and 6) reduced test time (columns 8 and 9) and reduction in items to test (columns 10 and 11). A reason for not answering questions, particularly about test time, is that several of the projects had not yet reached the test phase and hence it was difficult to report on defects and test time.

Outcomes for embedded and nonembedded systems were similar. Respondents for both system types reported noticeable savings in labor hours, however, both also had some projects that required more labor, as expected from existing literature. However, for both types of system most projects did not realize savings in the number of defects or in the time required to test the systems, contradicting expectations from existing literature.

Table 4.7: Outcome Summary

Category	Labor			Defects			Test Time			Test Items		
Savings/Cost	T	E	N	T	E	N	T	E	N	T	E	N
>20% Savings	31	18	13	10	6	4	5	2	3	7	5	2
10-20% Savings	23	10	13	4	3	1	9	7	2	10	5	5
0-10% Savings	5	3	2	7	2	5	19	9	10	20	8	12
No Savings	9	5	4	32	17	15	23	9	14	33	19	14
0-10% Added Cost	1	1	0	5	1	4	0	0	0	0	0	0
10-20% Added Cost	0	0	0	0	0	0	3	1	2	4	1	3
>20% Added Cost	3	1	2	0	0	0	0	0	0	0	0	0
No Answer	6	3	3	20	12	8	19	13	6	4	3	1
Total	78	41	37	78	41	37	78	41	37	78	41	37

* E = Embedded, N = Nonembedded

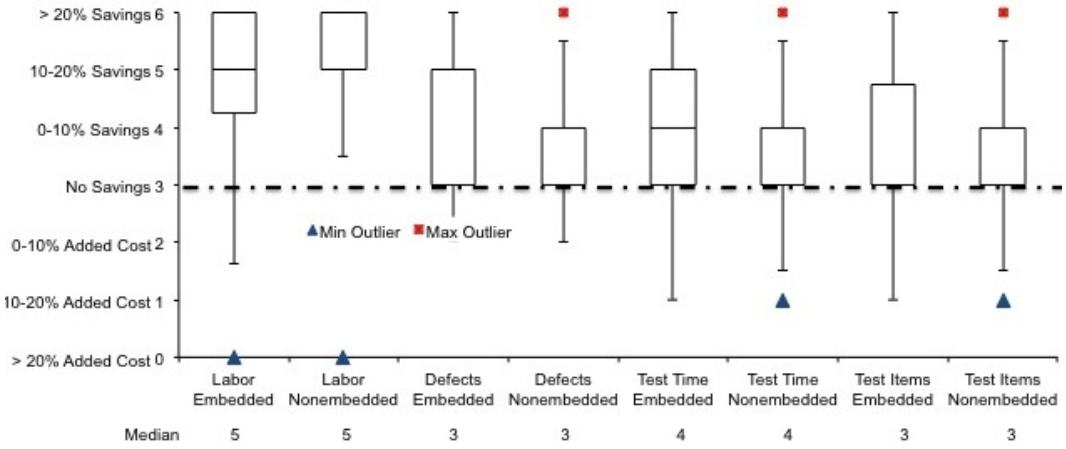


Figure 4.2: Outcomes

We then removed the responses "outcome unknown" and "no answer" and mapped the remaining answers onto a 7 point Likert scale for further box plot analysis. Figure 1 shows these box plots, comparing outcomes. The dashed line at point 3 (no savings) represents the break-even point.

Comparing the medians of outcomes for embedded and nonembedded systems shows that they are identical for all outcomes, but vary by outcome type. Labor savings has the most favorable outcome (a median of 10-20% savings) for both embedded and nonembedded systems. The median for defect reduction and test item reduction points to no

savings, while the median for test time reduction shows a 0 - 10% reduction. Generally, the variability of outcomes is higher for embedded systems as indicated by the size of the boxes.

When considering labor savings, nonembedded systems projects do a little better, since they don't show the negative results that embedded systems have (whiskers in box plot). The situation is reversed when considering defect reduction, test time and test items. Overall, though, 75% of the projects have positive outcomes or at least break even (all the boxes start at or above the break even line).

While the median outcomes did not differ between embedded and nonembedded systems, the spread in each outcome did show differences. Nonembedded systems showed a greater spread than embedded systems for labor and defects. There were, however, no outliers in nonembedded systems in any of the outcomes, whereas test time reduction and test item reduction showed outliers in both directions for nonembedded systems.

The visual results offered by the box plots suggest that reuse success may not be greatly impacted by whether a system is embedded or nonembedded. However, the variability in outcomes is much greater when the system is embedded. Larger variability in outcomes might be due to differences in project contexts that can vary more in embedded systems compared to nonembedded systems. An example of this would be hardware changes with large impacts on the software (e.g. going from 8 bit to 32 bit processors) versus changes that do not require large modifications for software (similar hardware). We will explore this more when we analyze free-form answers.

One major consideration in the decision to reuse a product is the risk that is either introduced or mitigated by use of the product. The risk is increased if there is a possibility of latent defects in the reused products. The risk is mitigated by the confidence that the reused product has already been proven. There did not appear to be much difference between respondents working on embedded systems from nonembedded systems when asked about whether reuse reduced risk. 31 of the subjects reporting on embedded systems projects thought reuse reduced risk, nine did not, and one did not respond. Conversely, 29 of the

subjects reporting on nonembedded system projects thought reuse reduced risk, while eight thought it did not. In addition, respondents were asked why they felt risk was or was not reduced. We will also address this when we analyze free-form answers.

RQ-4 Does reuse effectiveness vary with project size?

Finally, we attempted to determine whether the size of the project had an impact on reuse strategy or reuse success. However, as we analyzed the responses, it became clear that some respondents had reported their size in KLOC as requested in the survey, while others had reported size in LOC. There was no way to accurately determine which metric had been reported, thus this question could not be analyzed.

4.3.2 Hypothesis Testing

As the observations obtained from descriptive statistics were quite interesting, the question is whether they are statistically significant. We ran a MANOVA (Multivariate Analysis of Variance) test using SPSS for Surveys by IBM for questions 1, 2 and 3 in this section. [1] We report these results using the standard specified in [3]. MANOVA differs from ANOVA in that there are multiple dependent variables under consideration simultaneously. We report the F ratio, the significance p (we assume $\alpha = 0.5$) and $\eta^2$for the effect size (it indicates the approximate percentage of the variance accounted for with difference between the samples).

Statistical Analysis for RQ-1 - development approach variation based on the system type First we look at Wilks' λ. λ is a measure of whether the group means are equal across all dependent variables, here, the development approaches, not explained by differences in the level of the independent variable measures as a whole (i.e. embedded and nonembedded systems). In our case, Wilks' $\lambda = .924$, with an associated F of 1.179, whose p = .328, which is not significant at p <.05. We find that the partial η^2 associated with the main effect is .076 and the power to detect the main effect is only .396. Thus, we

[1] Our thanks to Dr. Andreas Stefik for performing the MANOVA calculations for this work.

have insufficient evidence to determine whether our main effects are different across the board. The MANOVA revealed a non-significant multivariate main effect for development approach. We cannot conclude that a system type is a major factor in determining the development approach overall.

We continue by looking at the specifics of the analysis, that is the impact on selection of development approaches based on whether the system is embedded or nonembedded, by looking at the Tests of Between-Subjects Effects. These tests behave like ANOVA, in that they test the effect of system type on the individual development approaches. These results are:

- AdHoc F(2, 76)=.112, p=.738, $\eta^2 = .001$
- Model based F(2, 76) = .993, p = .322, $\eta^2 = .013$
- Component based F(2, 76) = 3.798, p = .055, $\eta^2 = .048$
- COTS/GOTS F(2,76) = .631, p = .429, $\eta^2 = .008$
- Product Line F(2, 76) = 1.665, p = .201, $\eta^2 = .021$

We could not reject the Null hypothesis (i.e. no significant differences in development approaches in embedded vs nonembedded systems) for any of these measures, which is not surprising given the multivariate test previously reported. Since all the p-values are greater than .05, none of the p-values indicates significance. With that said, Component Based Development approaches significance (p=.055), but even if this result is significant, it accounts for less than 5% of the variance. None of the other development approaches show signs of being used by one system type more than the other. Hence while our descriptive statistics of Table 6 indicated differences, they are not statistically significant.

Statistical Analysis for RQ-2 - artifact variation based on the system type
We now look at the differences between embedded systems and nonembedded systems in the artifacts they select for reuse using MANOVA. These artifacts are requirements,

code, architecture, models, drawings, hardware, use cases, test products and already tested clusters, that is, groups of components that have been previously integrated and tested and shown to work together. MANOVA revealed a non-significant multivariate main effect for artifacts used, Wilks' $\lambda = .847$, F (2,76) = 1.365, p =.221, $\eta^2 = .153$. Power to detect the effect was .610. The null hypothesis cannot be rejected.

Looking at the Tests of Between Subjects Effects, we find:

- Requirements F(2, 76) = 3.131 p = .081, $\eta^2 = .040$
- Code F(2, 76) = 2.621 p = .110, $\eta^2 = .033$
- Architecture F(2, 76) = 4.065 p = .047, $\eta^2 = .051$
- Models F(2, 76) = .668 p = .416, $\eta^2 = .009$
- Drawings F(2, 76) = .107 p = .744, $\eta^2 = .001$
- Hardware F(2, 76) = 5.184 p = .026, $\eta^2 = .064$
- Use Cases F(2, 76) = 4.218 p = .043, $\eta^2 = .053$
- Test Products F(2, 76) = 6.269 p = .014, $\eta^2 = .076$
- Tested Clusters F(2, 76) = 3.897 p = .052, $\eta^2 = .049$

While our multivariate main effect for artifacts was not significant, some of the artifacts individually show significant results. Architecture, hardware, use cases and test products were significantly more likely to be used by embedded systems, with p values less than .05. Already tested clusters approached significance with the p value at .052 , in each case accounting for approximately 7.6% of the variance or less. This confirms our observations in Section 5.1.

Statistical Analysis for RQ-3 - outcome variation based on the system type
Based on our analysis of box plots, no significant differences can be expected, hence there was no point in running a MANOVA.

4.3.3 Principal Components Analysis

To augment our statistical analysis, we performed multiple Principal Components Analysis (PCA) to the survey response data of the survey questions listed in Table 4.8. The intent was to determine whether there were survey questions that tended to vary together, particularly questions related to project attributes and outcomes. [2]

PCA remains a popular method in software engineering to achieve reduction in large variable sets and to group variables into similarly-behaving subsets ([98, 99, 122, 123]). The predictors of software development project success factors based on project similarities in [18] use PCA to group projects with similarly varying characteristics. Project attributes and success indicators are evaluated to predict project outcome. For questions Q11 and Q14, the respondents were permitted to make multiple selections from fixed lists. The selections are identified as Q11a through Q11e, and Q14a through Q14i. The AHP approach for Q11 and Q14 follows [113]. We apply AHP to Q11 and Q14 independently. Table 4.9 shows our rubric for the assignment of integer values to survey response pairs for Q11 and Q14.

We first construct a decision hierarchy in which the possible selections are compared using criteria which we believe appropriately contribute to the ranking of the relative importance of the possible responses. We apply AHP to Q11 and Q14 separately, starting with Q11 then proceeding to Q14. Our goal in using AHP is to estimate the relative differences among reuse approaches in Q11, and to separately estimate the relative differences among reuse artifacts in Q14. Using AHP we determine a vector of five weights to be applied to the five parts to Q11, and a separate vector of nine weights to be applied to the nine parts of Q14. We then apply the weights to the response data for Q11 and Q14.

We choose AHP to prioritize answers to Q11 and Q14. Q11 addresses reuse strategies and includes five possible approaches from which the respondents may make one or more selections. Q14 addresses the reuse of nine different artifacts. Our experience in reuse strategies and artifacts suggests a relative order on the basis of their *evolution*. Similarly,

[2] Our thanks to Joe Lucente for performing the Principle Components Analysis section of this work.

Table 4.8: Survey Questions Used in PCA

	Abbreviated description	Basis for Inclusion in PCA
Q1	Embedded vs. Non-embedded	relates to RQ1 (relationship between embedded vs non embedded in reuse)
Q11	Development approach Q11a Component Based (cmp_bsd) Q11b Model Based (mod_bsd) Q11c Product Line Based (pl_bsd) Q11d COTS/GOTS Based (cg_bsd) Q11e Heritage Based* (her_ad) Q11f Ad hoc based	relates to RQ3 (relationship between development approachs and there effectiveness in embedded vs non embedded systems)
Q14	Product(s) reused Q14a Requirements (rqts) Q14b Code (code) Q14c Architecture (arch) Q14d Models (mod) Q14e Drawings (dwgs) Q14f Hardware (hdwr) Q14g Use Cases (ucs) Q14h Test products (testpr) Q14i Tested clusters (testdcl)	relates to RQ4 (relationship between artifacts and ease of reuse in embedded vs non embedded systems)
Q15	Reuse save labor? (sv_lb)	observable outcome of reuse in embedded vs non embedded systems
Q16	Fewer defects with reuse? (red_def)	observable outcome of reuse in embedded vs non embedded systems
Q17	Reuse reduce test time? (red_tm)	observable outcome of reuse in embedded vs non embedded systems
Q18	Reuse reduce items tested? (red_itms)	observable outcome of reuse in embedded vs non embedded systems
Q19	Reuse reduce risk? (red_rsk)	observable outcome of reuse in embedded vs non embedded systems

* responses for Heritage (Q11e) and Ad Hoc (Q11.f) were combined as Q11e Heritage/Ad Hoc in the PCA

Table 4.9: Scale of comparison for pairwise assignments

Relative Magnitude	Description
1	Equal
2	Weak
3	Slightly moderate
4	Moderate
5	Somewhat strong
6	Strong
7	Very strong
8	Exceptionally strong
9	Extreme

we observe differences in the level to which reuse artifacts have evolved. Using AHP, we derive a set of weights which we apply to the five possible responses in Q11 so that selections associated with a greater level of evolution are assigned a higher weight. We take the same approach to Q14. Our decision hierarchy for Q11 and Q14 contains response *frequency* as a second basis of comparison. We again assign numbers to relative differences between pair elements in the survey responses to Q11 and Q14, but this time we use frequency as the basis of comparison. In order to apply the sets of weights to achieve an overall relative prioritization reflective of our estimations on the basis of both evolution and frequency, we must assign weights to the two comparison parameters. We conclude a slight bias toward frequency is appropriate, and assign 60% of the weight to frequency and 40% to evolution. We refer to the comparison scale defined in Table 4.9 when making selections for each pairwise comparison ([113]).

Tables 4.10 and 4.11 show the results of our pairwise comparison of Q11 responses, evaluated on the basis of the relative difference in how *evolved* each reuse approach is in each pairwise comparison and on relative frequency of selection. Table 4.12 shows the weight vector for Q11 for both evolution and frequency.

Table 4.10: AHP Pairwise Selections Based on Relative Degree of Evolution of Reuse Artifacts (normalized) (Q11a-e)

	Q11.a	Q11.b	Q11.c	Q11.d	Q11.e	Row sum	Weights $\mathbf{W_{E_{Q11}}}$ (row avg.)	CM
Q11.a	0.490	0.511	0.516	0.444	0.381	2.342	0.468	5.096
Q11.b	0.245	0.255	0.258	0.296	0.286	1.340	0.268	5.065
Q11.c	0.122	0.128	0.129	0.148	0.190	0.718	0.144	5.036
Q11.d	0.082	0.064	0.065	0.074	0.095	0.379	0.076	5.022
Q11.e	0.061	0.043	0.032	0.037	0.048	0.221	0.044	5.011
Sum	1	1	1	1	1		1	

To obtain a weight vector for Q14, we repeat our AHP as described above. Table 4.13 shows the results for Q14. W_{Q11} and W_{Q14} were applied to survey responses for Q11 and Q14 respectively.

We conduct PCA on eleven different datasets from the survey data. Each dataset is comprised of a combination of two or more categories of survey response data. Recall,

Table 4.11: AHP Pairwise Selections Based on Relative Frequency of Selection in Survey Data (normalized) (Q11a-e)

	Q11a	Q11b	Q11c	Q11d	Q11e	Row sum	Weights $\mathbf{W_{E_{Q11}}}$ (row avg.)	CM
Q11a	0.077	0.067	0.059	0.077	0.088	0.367	0.073	5.015
Q11b	0.154	0.133	0.118	0.115	0.146	0.666	0.133	5.012
Q11c	0.154	0.133	0.118	0.115	0.109	0.630	0.126	5.018
Q11d	0.231	0.267	0.235	0.231	0.219	1.182	0.236	5.034
Q11e	0.385	0.4	0.471	0.462	0.438	2.155	0.431	5.046
Sum	1	1	1	1	1		1	

Table 4.12: AHP Synthesis of Weights and Consistency Metrics for Evolution and Frequency (Q11)

	Evolution $\omega_E = 0.4$			Frequency $\omega_F = 0.6$			
	Total	Weights $\mathbf{W_{E_{Q11}}}$	Consistency Metric (CM)	Total	Weights $\mathbf{W_{F_{Q11}}}$	Consistency Metric (CM)	Resulting weight vector*
Q11a comp based	2.342	0.468	5.096	0.367	0.073	5.015	0.231
Q11b model based	1.340	0.268	5.065	0.666	0.133	5.012	0.187
Q11c product line	0.718	0.144	5.036	0.630	0.126	5.018	0.133
Q11d COTS/GOTS	0.379	0.076	5.022	1.182	0.236	5.034	0.172
Q11e heritage/ ad hoc	0.211	0.044	5.011	2.155	0.431	5.046	0.276

* $\mathbf{W_{Q11}} = (\mathbf{W_{E_{Q11}}} \times \omega_E) + (\mathbf{W_{F_{Q11}}} \times \omega_F)$

the survey questions can be placed into four categories: 1) input (embedded vs. non-embedded), 2) reuse approach, 3) reuse artifacts, and 4) outcomes. We can construct datasets consisting of data from these categories, with the constraint that the dataset must include survey response data from at least two categories. The four categories result in $2^4 = 16$ unique combinations of categories. We define our eleven tests by eliminating the trivial case of no categories, and discard the four cases in which exactly one category appears. This leaves the eleven tests shown in Table 4.14, where a check mark in a column indicates inclusion of survey results from the category, in the test data.

We apply a separate PCA for each test pair. We note that test pairs can be selected such that exactly one category can be included and excluded between the two tests, while the

Table 4.13: AHP Synthesis of Weights and Consistency Metrics for Evolution and Frequency (Q14a-i)

	Evolution $\omega_E = 0.4$			Frequency $\omega_F = 0.6$			
	Total	Weights $\mathbf{W_{E_{Q14}}}$	Consistency Metric (CM)	Total	Weights $\mathbf{W_{F_{Q14}}}$	Consistency Metric (CM)	Resulting weight vector*
Q14a requirements	0.279	0.033	9.152	1.639	0.182	9.432	0.122
Q14b code	0.179	0.020	9.154	2.643	0.294	9.474	0.184
Q14c architecture	2.242	0.249	10.24	1.558	0.173	9.501	0.203
Q14d models	1.025	0.114	10.254	0.726	0.081	9.207	0.094
Q14e drawings	0.399	0.044	9.454	0.337	0.037	9.149	0.040
Q14f hardware	0.244	0.027	9.207	0.687	0.076	9.242	0.056
Q14g use cases	0.813	0.090	9.422	0.462	0.051	9.088	0.067
Q14h tested products	1.617	0.180	10.350	0.726	0.081	9.207	0.120
Q14i tested clusters	2.184	0.243	9.999	0.222	0.025	9.200	0.112

* $\mathbf{W_{Q14}} = (\mathbf{W_{E_{Q14}}} \times \omega_E) + (\mathbf{W_{F_{Q14}}} \times \omega_F)$

remaining categories are fixed. For example, test pair (All, A) differs only by outcomes; Test All includes all four categories but test A includes all categories except outcomes. Selection of test pairs which differ by exactly one category allows us to investigate the effect of resulting principal components in the presence and absence of the single category by which the two tests differ. Through this analysis we draw conclusions about the influence of the variable category on the covariance of the survey response data. The covariance is formally investigated through PCA. Table 4.14 shows the composition of data in Tests All through J, in terms of survey question categories.

We interpret results under the following assumptions::

1. If survey response relationships are found to exist in both tests in a test pair which differ by exactly one survey response category, we conclude that the presence of the category by which the tests differ has a diminishing effect on the survey response relationships (the principal components), the magnitude of which depends on the number of shared relationships between the tests. In other words, the greater the number of shared relationships, the smaller the effect of the single included/excluded category on the variance of the survey response relationships.

2. If the survey response relationships in one test are different than those in the other test (within the test pair), we conclude the effect the included/excluded category has on the survey response relationships is greater.

Each row r_i in M corresponds to survey question a_i in the original dataset. Each column c_j in M contains the vector of loadings for PC_j. As in [25], the Kaiser criterion is applied by selecting components with an eigenvalue (of the correlation matrix) greater than one and loadings of $|0.32|$. Table 4.15 shows the PCA loadings from the test in which survey data from all four categories is included. Tables 4.16 through 4.24 present the PCA loadings from tests A through J as defined in Table 4.14. The principal component loadings identified through the PCA loadings analysis described earlier are shown in bold in Tables 4.16 through 4.24. We develop survey response relationships directly from the flagged PCA loadings.

4.3.4 Analysis of PCA Results

First, we describe each test pair and include the results obtained from the PCAs in terms of the shared relationships (if there are any) between the test pairs. We then state the conclusions we draw from an analysis of the test results.

Table 4.14: PCA Tests Showing Survey Data Categories Included

	System Type	Reuse Approach	Reuse Artifact	Outcomes
All	✓	✓	✓	✓
A	✓	✓	✓	
B	✓	✓		
C	✓		✓	
D		✓	✓	
E			✓	✓
F		✓	✓	✓
G	✓	✓		✓
H	✓		✓	✓
I	✓			✓
J		✓		✓

]

4.3.5 Analysis of Pairs

Test pair (All, A): We compare Tables 4.15 and 4.16. This test pair investigates the effects on survey response relationships between system type (embedded/non-embedded), reuse approaches and reuse artifacts, when outcomes are included in the PCA vs. when outcomes are excluded. Test All includes data from all four categories. Test A includes data from all categories except outcomes.

Table 4.15: Test All: PCA Factor Loadings - All Survey Questions

Question	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
emb_non	-0.1319	-0.0602	0.2953	-0.2436	0.3052	-0.1856	**0.3406**	-0.1704
cmp_bsd	-0.1962	0.1329	-0.0934	**-0.4031**	-0.1455	0.3032	0.3361	-0.1083
mod_bsd	-0.2394	0.0497	-0.2143	**-0.4620**	-0.2334	0.1158	-0.0918	0.0983
pl_bsd	-0.2931	-0.1363	0.0642	-0.0765	0.163	0.0121	-0.2436	0.0593
cg_bsd	0.0091	-0.1803	-0.2144	-0.0657	-0.1818	**0.4718**	-0.3811	-0.4161
her_ad	-0.0046	-0.0477	**-0.4507**	0.1836	0.2115	0.1208	0.2773	-0.3124
rqts	-0.1676	**-0.4705**	0.0542	-0.0139	0.1618	0.0221	-0.1749	0.0444
code	-0.283	-0.0098	-0.0548	0.0581	**0.5013**	0.2761	0.0504	-0.0656
arch	-0.2514	-0.2696	-0.0427	0.0335	0.2278	0.0275	**-0.3376**	0.2735
mod	-0.2347	0.0413	**-0.3899**	-0.1815	-0.1134	-0.3751	0.0193	0.118
dwgs	-0.021	-0.2564	-0.0853	0.3437	**-0.3841**	0.1464	0.1747	0.329
hdwr	-0.1882	-0.2969	0.1827	0.0961	-0.2019	0.2908	**0.3629**	0.0382
ucs	-0.139	-0.2913	-0.3888	-0.1467	-0.0443	**-0.3936**	0.1347	-0.0109
tstpr	-0.1553	**-0.3965**	0.1701	0.1502	-0.1206	-0.1778	0.2427	-0.1958
tstdcl	-0.2072	0.0611	**0.3672**	-0.2855	-0.1369	0.0924	-0.0356	0.2632
sv_lb	-0.2904	0.3083	-0.0777	0.1917	0.2497	0.1093	0.1417	0.2254
red_def	-0.287	0.1417	0.0131	0.286	-0.1828	-0.2108	-0.2136	-0.0842
red_tm	-0.3302	0.1849	0.1303	0.1534	-0.1121	-0.1559	-0.1359	**-0.4096**
red_itms	**-0.3425**	0.1448	0.1773	0.1127	-0.2233	-0.0384	-0.0161	-0.2648
red_rsk	-0.243	0.2387	-0.1869	0.2747	-0.0538	0.1463	0.0935	0.2592

Results: Inclusion and exclusion of outcomes results in zero shared survey response relationships. All relationships belong to either test All or test A.

Conclusions: Since the relationships between reuse system type, approach and reuse artifacts vary, and none are shared between the two tests, we conclude outcomes is influential in survey response variance.

Table 4.16: Test A: PCA Factor Loadings - All Survey Questions - less outcomes

Question	PC1	PC2	PC3	PC4	PC5	PC6
emb_non	0.2616	0.0054	-0.2507	0.0251	-0.2588	**0.4624**
cmp_bsd	0.1264	**0.4701**	-0.0463	-0.1234	-0.4480	0.1819
mod_bsd	0.2270	**0.4333**	0.1519	0.3578	-0.1637	-0.2540
pl_bsd	0.2029	0.2371	-0.2430	-0.2345	-0.0878	-0.1600
cg_bsd	0.0590	0.1281	0.4341	-0.3216	-0.2392	**-0.4724**
her_ad	0.1268	0.0854	**0.5499**	-0.1678	0.0943	0.4671
rqts	**0.3771**	-0.0337	-0.0608	-0.1471	0.3050	-0.2044
code	0.2801	0.2081	-0.1055	**-0.3826**	0.2813	0.2451
arch	**0.3929**	-0.0046	-0.0487	-0.1559	0.3297	-0.2599
mod	0.2429	0.1715	0.2473	**0.5193**	0.1317	0.0765
dwgs	0.1345	**-0.4519**	0.2327	0.0226	-0.2859	-0.0994
hdwr	0.2949	-0.2699	-0.0003	-0.1462	**-0.4785**	-0.0160
ucs	**0.3460**	-0.1645	0.2544	0.2525	0.1141	0.1047
testpr	0.3211	**-0.3648**	-0.1175	-0.0149	-0.0516	0.0582
testdcl	0.2035	0.0162	**-0.3886**	0.3488	-0.0846	-0.1478

Test pair (All, G): We compare Tables 4.15 and 4.17. This test pair investigates the effects on survey response relationships between system type (embedded/non-embedded), reuse approaches and outcomes, when reuse artifacts are included in the PCA vs. when reuse artifacts are excluded. Test All includes data from all four categories. Test G includes data from all categories except reuse artifacts.

Table 4.17: Test G: PCA Factor Loadings - All Survey Questions - less reuse artifacts

Question	PC1	PC2	PC3	PC4
emb_non	-0.1165	-0.1005	**-0.6392**	0.2551
cmp_bsd	-0.2303	**-0.5318**	-0.0095	0.3868
mod_bsd	-0.2523	**-0.5819**	0.0628	0.0652
pl_bsd	-0.2874	-0.1537	-0.2320	-0.2239
cg_bsd	0.0417	-0.4388	0.4032	**-0.4680**
her_ad	0.0103	0.0136	**0.4444**	0.2854
sv_lb	-0.3764	0.2282	0.1558	**0.3993**
red_def	**-0.3862**	0.2183	0.0305	-0.3059
red_tm	**-0.4353**	0.1367	-0.0277	-0.2662
red_itms	**-0.4416**	0.0410	-0.0775	-0.2014
red_rsk	-0.3315	0.1788	**0.3762**	0.2570

Results: Inclusion and exclusion of reuse artifacts results in one shared survey response relationships.

- Survey responses which do not include a component-based reuse approach tend to also not indicate a model-based reuse approach.

Conclusions: Survey response relationships among system type, reuse approaches and outcomes does vary differently, for the most part, based on reuse artifacts. We conclude reuse artifacts in the survey responses is influential in survey response variance.

Test pair (All, H): We compare Tables 4.15 and 4.18. This test pair investigates the effects on survey response relationships between system type (embedded/non-embedded), reuse artifacts and outcomes, when reuse approach is included in the PCA vs. when reuse approach is excluded. Test All includes data from all four categories. Test H includes data from all categories except reuse approach.

Results: Inclusion and exclusion of reuse approach results in two shared survey response relationships.

- Survey responses which do not specify requirements as reuse artifacts also tend to not specify selection of tested products as reuse artifacts.
- Survey responses which do not specify models as reuse artifacts tend to specify tested clusters as reuse artifacts.

Conclusions: Since two out of five relationships in test H are also found in test All, we conclude that system type (embedded/non-embedded), reuse artifacts and outcomes are somewhat independent of reuse approach.

Test pair (All, F): We compare Tables 4.15 and 4.19. This test pair investigates the effects on survey response relationships between reuse approach, reuse artifacts and

Table 4.18: Test H: PCA Factor Loadings - All Survey Questions - less reuse approach

Question	PC1	PC2	PC3	PC4	PC5
emb_non	-0.1250	-0.1141	**0.3539**	-0.3449	0.2045
rqts	-0.1798	**-0.4649**	0.0337	-0.1657	-0.1240
code	-0.3053	0.0067	0.0855	-0.3533	**-0.4443**
arch	-0.2706	-0.2465	-0.1034	-0.2219	-0.2700
mod	-0.2305	0.0206	**-0.4988**	-0.1834	0.2924
dwgs	-0.0395	-0.2541	-0.1773	**0.5929**	-0.2241
hdwr	-0.2056	**-0.3264**	0.2409	0.3190	-0.1277
ucs	-0.1336	-0.3154	**-0.4876**	-0.1896	0.2735
tstpr	-0.2004	**-0.4335**	0.0929	0.1489	0.1493
tstdcl	-0.1997	0.0136	**0.4147**	-0.0338	0.1475
sv_lb	-0.3300	0.3134	-0.0487	-0.0931	**-0.3375**
red_def	**-0.3312**	0.1712	-0.1064	0.2380	0.1884
red_tm	**-0.3794**	0.1987	0.1201	0.0788	0.3253
red_itms	**-0.3827**	0.1384	0.1483	0.1852	0.2337
red_rsk	-0.2861	0.2543	-0.2211	0.1659	-0.3029

outcomes, when system type (embedded/non-embedded) is included in the PCA vs. when system type is excluded. Test All includes data from all four categories. Test F includes data from all categories except system type.

Results: Inclusion and exclusion of system type results in zero shared survey response relationships between Test All and Test F.

Conclusions: Since there are no shared relationships between the two tests, we conclude system type influences variance in the survey responses.

Test pair (A, B): We compare Tables 4.16 and 4.20.This test pair investigates the difference in survey response relationships with and without the specification of reuse artifacts, but independent of outcomes. Test A includes data from system type, reuse approach and reuse artifacts. Test B includes data from system type and reuse approach. Neither test contains data from outcomes.

Results: Inclusion and exclusion of reuse artifacts results in one shared survey response relationships between Test A and Test B.

Table 4.19: Test F: PCA Factor Loadings - All Survey Questions - less input

Question	PC1	PC2	PC3	PC4	PC5	PC6	PC7
cmp_bsd	0.1935	-0.1363	0.2452	-0.3382	0.1163	**-0.4311**	0.0887
mod_bsd	0.2439	-0.0443	0.3422	**-0.4269**	0.0879	-0.0710	-0.0987
pl_bsd	0.2908	0.1347	-0.0302	-0.0837	-0.2487	0.1255	-0.1224
cg_bsd	-0.0005	0.1929	0.1660	-0.1119	-0.0897	-0.2653	**-0.7408**
her_ad	0.0094	0.0590	0.3738	**0.4176**	-0.0252	-0.2416	-0.0313
rqts	0.1668	**0.4716**	-0.0602	-0.0452	-0.2459	0.0929	0.0543
code	0.2812	0.0094	0.0644	0.1826	**-0.4900**	-0.2139	0.1050
arch	0.2541	0.2750	0.0119	0.0370	**-0.3363**	0.1644	-0.0544
mod	0.2408	-0.0310	**0.4254**	-0.0275	0.2332	0.3040	0.1311
dwgs	0.0279	0.2686	-0.0963	0.2420	**0.4094**	-0.2674	-0.0968
hdwr	0.1870	0.2964	-0.2277	-0.0313	0.1888	**-0.4383**	0.1807
ucs	0.1398	0.2983	**0.4237**	0.0397	0.2300	0.2697	0.1840
tstpr	0.1504	**0.3933**	-0.2165	0.0738	0.2378	0.0306	0.1761
tstdcl	0.2033	-0.0709	-0.2508	**-0.4570**	-0.0343	-0.0455	0.2056
sv_lb	0.2958	-0.3025	0.0173	0.2624	-0.1792	-0.1381	0.2466
red_def	0.2939	-0.1328	-0.1455	0.1986	0.1977	0.2499	-0.3118
red_tm	**0.3336**	-0.1820	-0.1923	0.0579	0.1092	0.1832	-0.2422
red_itms	**0.3467**	-0.1419	-0.2330	-0.0296	0.1889	0.0197	-0.1310
red_rsk	0.2538	-0.2247	0.0516	0.2900	0.0867	-0.1897	0.0233

- Survey responses which indicate a component-based reuse approach also tend to specify a model-based reuse approach (independent of outcome)

Conclusions: Survey response relationships generally vary amongst inputs and reuse approaches (independent of outcome). Only one shared relationship does not allow us to conclude inputs and reuse approaches vary independent of reuse artifacts.

Test pair (A, C): We compare Tables 4.16 and 4.21. This test pair investigates the difference in survey response relationships with and without the specification of reuse artifacts, but independent of outcomes. Test A includes data from system type, reuse approach and reuse artifacts. Test B includes data from system type and reuse approach. Neither test contains data from outcomes.

Table 4.20: Test B: PCA Factor Loadings - All Survey Questions - less reuse artifacts less outcomes

Question	PC1	PC2
emb_non	0.3183	**-0.4443**
cmp_bsd	**0.5757**	-0.0849
mod_bsd	**0.5502**	0.0801
pl_bsd	**0.3868**	-0.3404
cg_bsd	0.2689	**0.6308**
her_ad	0.2063	**0.5246**

Table 4.21: Test C: PCA Factor Loadings - All Survey Questions - less reuse approaches less outcomes

Question	PC1	PC2	PC3	PC4
emb_non	0.2739	0.0235	0.1095	**-0.5531**
rqts	**0.3960**	0.2461	0.1321	0.2293
code	0.2653	**0.5030**	0.2582	0.0917
arch	**0.4113**	0.2925	0.0767	0.2797
mod	0.2154	0.0551	**-0.7673**	0.0167
dwgs	0.1882	**-0.5981**	0.0714	0.2274
hdwr	0.3292	**-0.3928**	0.2792	-0.1024
ucs	0.3762	-0.1334	**-0.4310**	0.1548
testpr	**0.3851**	-0.2470	0.1672	0.0033
testdcl	0.2205	0.0804	-0.1117	**-0.6844**

Results: Inclusion and exclusion of reuse approach results in one shared survey response relationships between Test A and Test C.

- Survey responses which indicate selection of code as a reuse artifact tend to not specify a drawings as a reuse artifact (independent of outcome)

Conclusions: Survey response relationships generally vary amongst inputs and reuse artifacts (independent of outcome). Only one shared relationship does not allow us to conclude inputs and reuse artifacts vary independent of reuse approaches.

Test pair (A, D): We compare Tables 4.16 and 4.22. This test pair is selected to investigates the difference in survey response relationships with and without the specification of system type (embedded vs. non-embedded), but independent of outcomes. Test A includes data from system type, reuse approach and reuse artifacts. Test D includes data from reuse approach and reuse approach. Neither test contains data from outcomes.

Results: This test pair exposed the greatest number of shared relationships between survey responses. Nine shared survey response relationships are observed between Test A and Test D.

- Survey responses which specify a component-based reuse approach
  - also tend to specify a model-based reuse approach.
  - tend to not specify drawings as a reuse artifact
  - tend to not specify tested products as a reuse artifact
- Survey responses which specify a model-based reuse approach
  - tend to not specify drawings as a reuse artifact
  - tend to not specify tested products as a reuse artifact
- Survey responses which do not specify drawings as a reuse artifact also tend to not specify tested products as a reuse artifact
- Survey responses which specify a heritage/ad hoc reuse approach tend to not specify tested clusters as a reuse artifact
- Survey responses which do not specify code as a reuse artifact tend to specify models as a reuse artifact
- Survey responses which do not specify hardware as a reuse artifact tend to vary independently (i.e., hardware appears as a single factor in a principal component)

Conclusions: Because of the relatively large number of shared relationships between Test A and Test D, we conclude that system type has very little in-

fluence on the relationships between reuse approach and reuse artifacts, in the absence of outcome data. System type does not appear in any of the shared relationships in Test A. A comparison of the results of test pair (All, F) is of interest, since that test pair includes outcomes where test pair (A, D) does not. In the reuslts from (All, F) we concluded system type *does* matter in the variance of relationships between reuse approach, reuse artifacts and outcomes. In (A, D), where the absence of outcomes is the only difference compared to (All, F), system type does not influence the shared relationships. For this reason, we further conclude that outcomes must be considered when analysing the survey data.

Table 4.22: Test D: PCA Factor Loadings - All Survey Questions - less system type less outcomes

Question	PC1	PC2	PC3	PC4	PC5
cmp_bsd	-0.1124	**0.4783**	-0.0677	0.1099	-0.4064
mod_bsd	-0.2327	**0.4600**	0.2600	-0.2562	-0.1946
pl_bsd	-0.2077	0.1925	**-0.3876**	0.0629	-0.1808
cg_bsd	-0.0761	0.1710	0.2169	**0.4753**	-0.2658
her_ad	-0.1405	0.1316	0.3927	**0.4298**	0.2657
rqts	**-0.3950**	-0.0664	-0.2108	0.0510	0.2286
code	-0.2912	0.1995	**-0.3219**	0.2598	0.2651
arch	**-0.4108**	-0.0376	-0.2108	0.0616	0.2503
mod	-0.2565	0.2117	**0.4194**	-0.3507	0.1602
dwgs	-0.1489	**-0.4327**	0.2613	0.1072	-0.2877
hdwr	-0.3012	-0.2681	-0.0135	0.1458	**-0.5018**
ucs	**-0.3625**	-0.1428	0.3303	-0.1057	0.1777
tstpr	-0.3239	**-0.3256**	-0.0250	-0.0372	-0.1004
tstdcl	-0.1999	-0.0033	-0.1726	**-0.5153**	-0.1895

Test pair (F, E) and Test pair (F, J): We compare Tables 4.19, 4.23, and 4.24. These two test pairs are selected to compare the difference in survey response relationships with and without reuse approach (F, e), and with and without reuse artifacts (F, J), with each test pair independent of system type.

Table 4.23: Test E: PCA Factor Loadings - All Survey Questions - less system type less reuse approach

Question	PC1	PC2	PC3	PC4	PC5
rqts	-0.1879	**0.4661**	-0.0151	0.3034	-0.0909
code	-0.2955	-0.0108	0.0664	**0.5011**	0.2170
arch	-0.2790	0.2564	0.1252	**0.3517**	0.1008
mod	-0.2175	-0.0048	**0.5658**	-0.1948	-0.1709
dwgs	-0.0540	0.2872	-0.1591	-0.4458	**0.5262**
hdwr	-0.2113	0.3331	**-0.3895**	-0.1027	0.1448
ucs	-0.1360	0.3333	**0.5353**	-0.1515	-0.1278
tstpr	-0.1840	**0.4260**	-0.1122	-0.1892	-0.1407
tstdcl	-0.1991	-0.0351	-0.3508	0.1597	**-0.3670**
sv_lb	**-0.3282**	-0.2852	0.0381	0.2123	0.3270
red_def	**-0.3432**	-0.1680	0.0073	-0.3035	-0.0213
red_tm	**-0.3872**	-0.2122	-0.1020	-0.1417	-0.3204
red_itms	**-0.3913**	-0.1444	-0.1896	-0.1807	-0.2069
red_rsk	-0.2900	-0.2329	0.1120	-0.1083	**0.4287**

Table 4.24: Test J: PCA Factor Loadings - All Survey Questions - less system type less reuse artifacts

Question	PC1	PC2	PC3	PC4
cmp_bsd	0.2250	**-0.5236**	0.2437	-0.3471
mod_bsd	0.2543	**-0.5878**	0.0152	-0.1747
pl_bsd	0.2811	-0.1284	-0.2970	0.1939
cg_bsd	-0.0315	-0.4854	-0.1069	**0.5790**
her_ad	-0.0051	-0.0272	**0.5978**	0.5962
sv_lb	0.3808	0.2124	**0.4058**	-0.1555
red_def	**0.3918**	0.2085	-0.2048	0.2407
red_tm	**0.4375**	0.1351	-0.2087	0.1893
red_itms	**0.4443**	0.0439	-0.2329	-0.0480
red_rsk	0.3426	0.1365	**0.4231**	-0.0212

Results: Test pair (F, E) results in two shared survey response relationships.

- The selection of requirements for reuse varies with the selection of tested products
- The selection of models as a reuse artifact varies with the selection of use cases.

Results: Test pair (F, J) results in one shared survey response relationship.

- A reduction of items to be tested results in a reduction of test time.

Conclusions: Although there are some shared survey response relationships in test pairs (F, E) and (F, J), there is still general indication from the relatively small number of shared relationships that reuse approach and reuse artifacts do contribute to variance in the survey responses.

Test pair (F, D) and Test pair (H, C): We compare Tables 4.19, 4.22, 4.18,and 4.21. These two test pair are selected to compare the difference in survey response relationships with and without outcomes, independent of system type in (F, D), and independent of reuse approach (H, C).

Results: Test pair (F, D) results in one shared survey response relationship.

- Survey responses which specify a heritage/ad hoc reuse approach tend to not specify tested clusters as a reuse artifact.

Results: Test pair (H, C) results in one shared survey response relationship.

- Survey responses which do not indicate reuse of models also tend to not specify the reuse of use cases.

Conclusions: There is general indication from the small number of shared relationships in both Test (F, D) and (H, C) that inclusion/exclusion of outcome data in the PCA for (F, D) does influence variance in the relationships between reuse approach and reuse artifacts. Similarly, inclusion and exclusion of outcomes also influences variance between system type and reuse approaches.

Test pair (H, I): We compare Tables 4.18 and 4.25. This test pair compares the difference in survey response relationships with and without reuse artifacts, independent

of reuse approach. Test H includes data from system type, reuse artifacts and outcomes. Test I contains system type and outcomes.

Table 4.25: Test I: PCA Factor Loadings - All Survey Questions - less reuse approach less reuse artifacts

Question	PC1	PC2
emb_non	-0.0742	**0.7513**
sv_lb	**-0.4208**	-0.2818
red_def	**-0.4413**	0.0722
red_tm	**-0.4921**	0.2530
red_itms	**-0.4789**	0.2062
red_rsk	-0.3888	**-0.4944**

Results: Test pair (H, I) results in three shared survey response relationships.

- Outcomes resulting in a reduction of defects vary with a reduction in test time.
- Outcomes resulting in a reduction of defects vary with a reduction in the number of items to be tested.
- Outcomes resulting in a reduction of test time vary with a reduction in the number of items to be tested.

Conclusions: Although system type is included in both test H and test I, none of the three shared relationships include system type. In fact, all three components in all relationships are outcomes. This suggests outputs vary with one another 1) independent of inputs, and 2) independent of reuse artifact.

Test pair (H, E): We compare Tables 4.18 and 4.23. This test pair compares the difference in survey response relationships with and without system type, independent of reuse approach. Test H includes data from system type, reuse artifacts and outcomes. Test E contains data from reuse artifact outcomes.

Results: Test pair (H, E) results in three shared survey response relationships. These are the same three as were found in test (H, I).

- Outcomes resulting in a reduction of defects vary with a reduction in test time.
- Outcomes resulting in a reduction of defects vary with a reduction in the number of items to be tested.
- Outcomes resulting in a reduction of test time vary with a reduction in the number of items to be tested.

Conclusions: Some evidence exists that system type does not influence the relationship between the specification of reuse artifacts and outcome, in the absence of reuse approach.

Table 4.26: Number of Principal Components Shared Between Test Pairs

	All	A	B	C	D	E	F	G	H	I	J
All	-			1				1	1		1
A		-	1	1	**9**		1		1		
B			-		**2**			1			
C				-		1	1		1		
D					-		1				
E						-	**2**	**3**		**3**	
F							-				1
G								-	**3**	**3**	1
H									-		
I										-	
J											-

4.3.6 Summary of Results

The PCA applied to the survey data produced relationships between survey responses as shown in Tables 4.27 and 4.28. For each relationship we show a symbolic representation of

Table 4.27: Survey Response Relationships with Shared Principal Components

Relationship	All	Test A	Test B	Test C	Test D	Test E	Test F	Test G	Test H	Test I	Test J
rqts ↓↔ tstpr ↓	PC2								PC2		
mod ↓↔ tstdcl ↑	PC3								PC3		
cmp_bsd ↓↔ mod_bsd ↓	PC4							PC2			PC2
code ↑↔ dwgs ↓	PC5			PC2							
rqts ↑↔ arch ↑		PC1		PC1							
cmp_bsd ↑↔ mod_bsd ↑		PC2	PC1		PC2						
cmp_bsd ↑↔ dwgs ↓		PC2			PC2						
cmp_bsd ↑↔ tstpr ↓		PC2			PC2						
mod_bsd ↑↔ dwgs ↓		PC2			PC2						
mod_bsd ↑↔ tstpr ↓		PC2			PC2						
dwgs ↓↔ tstpr ↓		PC2			PC2						
her_ad ↑↔ tstdcl ↓		PC3			PC4		PC4				
code ↓↔ mod ↑		PC4			PC3						
hdwr ↓		PC5			PC5						
emb_non ↓↔ her_ad ↑			PC2					PC3			
cg_bsd ↑↔ her_ad ↑			PC2		PC4						
rqts ↑↔ tstpr ↑				PC1		PC2	PC2				
mod ↓↔ ucs ↓				PC3					PC3		
sv_lb ↓↔ red_def ↓						PC1				PC1	
sv_lb ↓↔ red_tm ↓						PC1				PC1	
sv_lb ↓↔ red_itms ↓						PC1				PC1	
red_def ↓↔ red_tm ↓						PC1		PC1	PC1	PC1	
red_def ↓↔ red_itms ↓						PC1		PC1	PC1	PC1	
red_tm ↓↔ red_itms ↓						PC1		PC1	PC1	PC1	
mod ↑↔ ucs ↑						PC3	PC3				
red_tm ↑↔ red_itms ↑							PC1				PC1

the relationship to supplement the contextual interpretations listed in the results section of each test pair described earlier. The following symbolic notation is used to describe behavioral relationships between survey response pairs:

$$q_u\{\uparrow \mid \downarrow\} \Rightarrow q_v\{\uparrow \mid \downarrow\} \tag{4.3.1}$$

The notation is interpreted as "in survey response pair (q_u, q_v) for $u \neq v$, as q_u $\{increases \uparrow \ or \ decreases \downarrow\}$, q_v $\{increases \uparrow \ or \ decreases \downarrow\}$. The attribute relationship

$$her\_adhoc \uparrow \ \Rightarrow mod \downarrow \tag{4.3.2}$$

means "survey responses which indicate a reuse approach is heritage/adhoc tend to not indicate models are reused." Relationships are interpreted in the context of the behavioral tendency of all survey responses included in the analysis. All survey response pairs for each factor in our PCA are annotated in this manner.

Table 4.28: Survey Response Relationships with Single Principal Components

Relationship		All	Test A	Test B	Test C	Test D	Test E	Test F	Test G	Test H	Test I	Test J
red_itms ↓		PC1										
her_ad ↓↔	mod ↓	PC3										
her_ad ↓↔	tstdcl ↑	PC3										
cg_bsd ↑↔	ucs ↓	PC6										
emb_non ↑↔	arch ↓	PC7										
emb_non ↑↔	hdwr ↑	PC7										
arch ↓↔	hdwr ↓	PC7										
red_tm ↓		PC8										
rqts ↑↔	ucs ↑		PC1									
arch ↑↔	ucs ↑		PC1									
emb_non ↑↔	cg_bsd ↓		PC6									
cmp_bsd ↑↔	pl_bsd ↑			PC1								
mod_bsd ↑↔	pl_bsd ↑			PC1								
emb_non ↓↔	cg_bsd ↑			PC2								
arch ↑↔	tstpr ↑				PC1							
code ↑↔	hdwr ↓				PC2							
dwgs ↓↔	hdwr ↓				PC2							
emb_non ↓↔	tstdcl ↓				PC4							
rqts ↓↔	arch ↓					PC1						
rqts ↓↔	ucs ↓					PC1						
arch ↓↔	ucs ↓					PC1						
pl_bsd ↓↔	code ↓					PC3						
pl_bsd ↓↔	mod ↑					PC3						
cg_bsd ↑↔	tstdcl ↓					PC4						
mod ↑↔	hdwr ↓						PC3					
hdwr ↓↔	ucs ↑						PC3					
code ↑↔	arch ↑						PC4					

Continued on next page

Table 4.28 – continued from previous page

Relationship	All	Test A	Test B	Test C	Test D	Test E	Test F	Test G	Test H	Test I	Test J
dwgs ↑↔ tstdcl ↓						PC5					
dwgs ↑↔ red_rsk ↑						PC5					
tstdcl ↓↔ red_rsk ↑						PC5					
mod_bsd ↓↔ her_ad ↑							PC4				
mod_bsd ↓↔ tstdcl ↓							PC4				
code ↓↔ arch ↓							PC5				
code ↓↔ dwgs ↑							PC5				
arch ↓↔ dwgs ↑							PC5				
cmp_bsd ↓↔ hdwr ↓							PC6				
cg_bsd ↓							PC7				
emb_non ↓↔ red_rsk ↑								PC3			
her_ad ↑↔ red_rsk ↑								PC3			
cg_bsd ↓↔ sv_lb ↑								PC4			
rqts ↓↔ hdwr ↓									PC2		
hdwr ↓↔ tstpr ↓									PC2		
emb_non ↑↔ mod ↓									PC3		
emb_non ↑↔ ucs ↓									PC3		
emb_non ↑↔ tstdcl ↑									PC3		
ucs ↓↔ tstdcl ↑									PC3		
dwgs ↑									PC4		
code ↓↔ sv_lb ↓									PC5		
emb_non ↑↔ red_rsk ↓										PC2	
red_def ↑↔ red_tm ↑											PC1
red_def ↑↔red_itms ↑											PC1
her_ad ↑↔ sv_lb ↑											PC3
her_ad ↑↔ red_rsk ↑											PC3
sv_lb ↑↔ red_rsk ↑											PC3
cg_bsd ↑											PC4

4.3.7 Discussion of Qualitative Results

We turn to the free form comments to look for reasons for the differences and commonalities in embedded and nonembedded system projects that employed reuse. For this discussion, comments by survey respondents (referred to as R followed by their id number) try to shed light on reasons for success and failure.

Development Approach Developers of embedded systems cited benefits from using component based reuse. One benefit was not having to reengineer or rewrite the test sets

(R58). The fact that the technical solution was clear and already deployed in a prior project was cited by four (R15, R38, R50 and R74). Experience of the engineer and pedigree of the products (R24), reuse of a proven methodology (R28), and developer confidence in the products (R28) were mentioned. Not touching common code or infrastructure was important to embedded systems developers using component based development as part of their solution (R34, R66). This could explain the fact that over half of the embedded systems used a component based development approach, as shown in Table 4.5.

Reasons for a model-based approach include that models had been used in other projects that reduce their risk (R48). These comments were echoed by respondents reporting on nonembedded systems. For both component based and ad hoc reuse, prior deployment and pedigree of the products (R69, R70, R37, R48, R34, R24) and developer confidence in trusted library components (R79) was cited. Reasons in favor of product line reuse in nonembedded systems included risk reduction (R71), a good match of the reused software with the new project (R23), and, over time, the development of reusable expertise (R52). No reasons were provided for the benefits in reuse of COTS/GOTS.

Artifact Reuse The quantitative analysis showed differences in the artifacts used between developers of embedded and nonembedded systems. From the respondent comments, we can see that both embedded and nonembedded systems practitioners mentioned software, code, components and test. However, while hardware was a major consideration for embedded systems, it was not mentioned by nonembedded systems developers.

Presumably, reuse in embedded systems is facilitated when code does not have to be adapted to new hardware. This is echoed in comments by respondent (R58) as a reason for success and a reason for reuse failure when hardware does change. Well known and documented circuits (R33), stable hardware (R8), and running the software on known platforms (R43, R72), not having to reengineer or rewrite test sets (R58) reduced risk as well. Again, this supports the conclusion that changes to the platform impact reusability, accounting for the high reuse of both code and hardware shown in Table 4.6. (R51) cites

reuse of code and architecture artifacts together as the reason for reuse success in embedded systems (with regard to development and testing costs), which is also supported by (R31) and (R33). (R56) points out that in cases where code reuse is feasible for embedded systems, this allows for higher efficiency and lower risk. (R62), (R83) and (R57) report reuse of code, component interfaces, components, requirements and test procedures as reasons for reuse success related to quality. Risk was believed to be reduced by reuse of code because it had been tested multiple times and the fact that a large percentage of the code was already integrated (R68), (R25), and because the of the use of proven equipment, designs and requirements (R71). Note that if the code is already tested, it must not have been changed, because if it was changed it would have to be tested again. Proven equipment, designs and requirements also imply minimal change. Proven models, design and code elements (R42), and already deployed projects based on the reused models (R42, R64, R81) were also mentioned. This could explain both the fact that all artifacts were reused in embedded systems at a higher rate than in nonembedded systems and the high reuse rate of requirements, architecture, code, hardware and test products by embedded systems, both shown in Table 4.6.

Unsuccessful reuse in embedded systems was explained by not being able to use or "fit" the reuse artifact: architecture mismatch with third party code (R51), code or reuse plan mismatch (R15), lack of availability or obsolete test products (R32), overly complex software artifacts (R9), (R12),(R57), (R32), (R51), and problematic legacy software (R58). The close coupling between hardware platforms and software is one important reason given. Another reason is that the technical solution was proven in an environment where performance and timing are critical. The fact that the hardware and software had proven themselves and the likelihood of introducing new defects was reduced was an important consideration. Developers knew that these products worked well together. While these were also cited by developers of nonembedded systems developers, embedded systems developers clearly relied on the past success of their components for new efforts.

Reasons for Success/Failure In nonembedded systems, respondent comments related to successful artifact reuse cited good documentation (R25), (R69), a good fit of the reused software to the new use (R23), and availability of product-line artifacts (R71).

By contrast, mismatches of reused software with needs for the new project ((R23), (R82)), poor quality of code or design ((R23), (R52)), poor documentation of reusable artifacts ((R79), (R69), (R14), (R82)), and complexity of the code base (R69) were cited as reasons for increased cost or failure. Undocumented or latent problems were cited by ((R82), (R5)) as reasons for much higher testing efforts.

While all were likely to reuse code, the reuse of use cases, hardware, test products and tested clusters (where no nonembedded systems developers cited use) was much greater in embedded systems. In fact, there may be a link in the reuse of hardware, test products and test clusters. This makes sense, because the test products would have been written to test the software on the hardware, and the resulting tested product groupings would be trusted in the new system. Test products could use parameters for any new requirements rather than developing an entire new test suite, a benefit cited by embedded systems developers. This observation is consistent with the quantitative analysis results in Table 4.6.

For embedded systems, successful reuse resulted in fewer defects, as mistakes were seldom repeated (R78), successes led to 4 times productivity and 10 times quality (R38). The increase in productivity is reflected in Table 4.7 and the box plots. However, outcomes for quality in terms of defects are less supported by the data in that table and the box plots. The reasons for the difference in the comments and the data are not clear. (R31) points out risk reduction due to proven artifacts. (R62), (R83) and (R57) comment on reduced testing as a benefit of reuse. (R56) points out that successful reuse allows for a corporate knowledge base that facilitates training new hires. These comments are similar to what is known in existing literature.

For unsuccessful reuse in embedded systems, some of the reasons given include lack of software comprehension (R70), complexity of resulting software and associated maintenance and testong costs ((R9), (R12), (R57), (R32), (R51)) and obsolescence or decay

in legacy systems (R21), unrealistic expectations (R58) and lack of management support(R16).

Respondent comments related to successful reuse in nonembedded systems reported savings in effort ((R4), (R35)), reduced risk ((R4), (R35)), easier training of new hires (R35), reduced team size (R71), lower cost (R34), increased quality ((R34), (R79)). Overall, the reasons cited for success and failure of reuse in embedded vs nonembedded systems are similar.

Risk Reduction In embedded systems, developers stated that risk was considered reduced in software because of application stability (R8), software maturity, (R16), and trusted components, software and systems (R19). The respondents cited limiting the number of variants leading to less unique code (R12), and already tested items with little adaptation, (R54, R57, R62, R70) as reasons. Finally, in terms of overall risk reduction with regard to reuse, the fact that the technical solution was clear and already deployed in a prior project (R15, R38, R50, R74) was a major consideration. Maturity of existing library items (R67), and previously corrected defects in functional and interface requirements, component design and coding (R83) were also mentioned.

For nonembedded systems, risk reduction in the area of software and code (including data) included experience of the engineer and pedigree of the software products (R24), reuse of a proven methodology (R28) maturity of the code base (R45), and developer confidence (R82). Not touching common code or infrastructure was also mentioned (R34, R66). For test, developers cited the need to only perform regression tests (R13), only having to test for usability (R14), proven functionality (R20) and use of a common core for all test set software (R63). Schedule risk was also believed to be reduced. The risk of not completing the project on time was reduced by reusing software (R23) and an entire development effort was eliminated (R53, R75) General comments about reuse reducing risk included reduction in uncertainty (R52). Hence the reasons for risk reduction echo the benefits reported above.

4.4 Discussion of Results

4.4.1 Descriptive Statistics

The descriptive statistics indicated an observable difference in the development approach between embedded and nonembedded systems, with embedded systems more likely to use CBSE, and product line development, and nonembedded systems more likely to use ad hoc and COTS/GOTS. We notice that those development approaches favored by embedded systems were the ones not favored by nonembedded systems and vice versa. Reasons respondents gave for this difference included the reuse of existing code already written to hardware, tests already written for those hardware/software components and the frequency in which clusters of hardware/software components had already been integrated and tested together. Nonembedded systems indicated a preference for MBSE because the fact that models were already proven in earlier projects, thus reducing risk.

The differences in artifacts reused seem to reflect the preference of development approach. Reuse levels for all artifacts was observably higher in embedded systems than in nonembedded systems. This was especially true for reuse of requirements, architecture, use cases, hardware, test products and test clusters. In the statistical analysis, this difference was significant for architecture, hardware, use cases, and test products, and nearly significant for already tested clusters. Again, this was explained by requirements stability, the benefit of running software on a known platform, hardware stability, and the fact that the components had been integrated, tested and deployed successfully on earlier projects, reducing risk. Because the hardware/software components did not change, the test products were still valid.

The box plots showed no observable difference in the median of outcomes between embedded systems and nonembedded systems. So, while there was difference in the development approaches and reuse artifacts, there was not a difference in the success of reuse itself. It is also interesting to notice that in terms of items to be tested and in defects, the median responses indicated no savings. It is also interesting to observe how many re-

sponses indicated additional cost attributed to reuse, sometimes that cost was surprisingly high. Only savings in labor showed a median of over 10%. Some of the observations of the respondents suggested that either the reuse was not as reusable as had been expected or that the reuse required developers experienced with the products in order to be effective. It was suggested that, in an embedded system, changing the hardware made it necessary to substantially change the software. This is supported by the respondents' comments about reasons for reuse success that the hardware and software had already been successfully integrated and tested.

4.4.2 Quantitative Statistics

Based on the MANOVA, there was little difference in the outcomes between embedded systems and nonembedded systems. So, while there was a difference in the development approaches and reuse artifacts, there was not a difference in the success of reuse itself. The MANOVA indicated that there was a difference in the development approaches in embedded systems vs the development approaches in nonembedded systems. Embedded systems were significantly more likely than nonembedded systems to use a heritage/legacy approach. There was also a significantly higher likelihood to reuse the hardware, test products and test clusters in embedded systems, and to use them together. It makes sense that hardware, test products and test clusters would be reused in embedded systems, because embedded software is often optimized to the processor. We noticed that several subjects commented that the greatest benefit of reuse in embedded systems was that the software was already tested against the hardware. This begs the question: Is it useful to reuse code in embedded systems if the hardware changes?

4.4.3 PCA

We found that a reduction in the number of items that requires testing through reuse varies independently from system type, reuse approach and reuse artifact. In fact, reuse in general reduces items to be tested, regardless of system type, reuse approach or reuse

artifact. We also found that reuse results in a reduction in test time regardless of reuse approach or artifact. We found that the use of heritage/ad hoc reuse approach results in less use of models as reuse artifacts, and greater use of tested clusters. Less selection of models as reuse artifacts would then be expected with less heritage/ad hoc. We found that a COTG/GOTS reuse approach leads to less reuse of use cases. Through our knowledge of system reuse, a selection of COTS/GOTS usually would concentrate more on the reuse of more developed artifacts such as tested products and less on existing use cases. Two interesting relationships exist between system type, architecture and hardware. We observed a three-way relationship where an embedded system type leads to less architecture reuse and more hardware reuse.

A summary of the significant findings from all tests is:

- A reduction of items to be tested is not associated with any particular system type, reuse artifact or reuse approach when outcomes are excluded.
- A reduction of test time is not associated with any particular system type, reuse artifact or reuse approach.
- Experience with embedded systems tends to avoid a COTS/GOTS as a reuse approach.
- Risk is reduced for nonembedded systems.
- Risk is reduced through a heritage/ad hoc reuse approach.
- Avoidance of a COTS/GOTS based reuse approach saves labor.
- More often than not, reuse approach and reuse artifact vary together but independent of both system type and outcome.
- Embedded systems reuse fewer models and use cases, but more tested clusters when analyzed without reuse approaches.

4.5 Threats to Validity

Because this is a mixed methods study, we need to look at threats to validity both from a quantitative and qualitative perspective.

4.5.1 Quantitative Threats to Validity

Wohlin et al [129] name four quantitative threats to validity: conclusion validity, internal validity, construct validity and external validity.

Conclusion Validity is concerned with the relationship between the treatment and the outcome. One of our threats to conclusion validity is the low statistical power of these tests. This is the result of a fairly low sampling and the large number of variables. We recommend repeating this study with more observations and the removal of the variables that indicated little impact on the treatments.

Internal Validity is concerned with whether there is in fact a causal relationship and not influenced by a factor that has not been measured. A threat to conclusion validity in this study is the self selection in this survey. Subjects may have chosen to take the survey because they had an agenda related to reuse. This has been mitigated by the observation that the answers indicate a nearly equal number of those who have issues with reuse and those who are enthusiastic about reuse.

Construct Validity is concerned with whether the treatment is in fact related to the cause, and that the outcome does in fact reflect the construct. One construct threat to validity in this experiment is the interaction of the different variables. For example, while we were intending to study the differences between embedded and nonembedded systems, the dependent variables of development approach may have had an impact on the artifacts selected. We mitigate this threat by studying the variable types separately.

External Validity is concerned with the ability to extend conclusions outside of the experiment. One major consideration is that many of the projects reported on have been ongoing for many years. Another is that the projects are either very large systems themselves, or research and development to be inserted into very large systems. Whether smaller projects would experience different results is not clear. Finally, these systems are from one domain. Whether different domains would experience the same phenomenon is not clear.

4.5.2 Qualitative threats to validity

Since some of the information collected is qualitative, we assess our approach with respect to: descriptive validity, interpretive validity, theoretical validity, generalizability, and evaluative validity [90].

Descriptive Validity relates to the quality of what the subject reports having seen, heard or observed. Here, not only are these observations subjective, since the projects are so large the respondents may see only a small part of the effort. Their observations may not reflect the whole project.

Interpretive Validity is concerned with what objects, events, and observations mean to the subjects. In our case, impartiality of reporting is uncertain.

Theoretical Validity refers to an account's validity as a theory of some phenomenon. It depends on the validity of the construct of the experiment and on the validity of the interpretation or explanation of the observations. It also depends on whether there is consensus within the community about terms used to describe context, events and outcomes. With the number of similar but not identical metrics presented, and ways of measuring success, there is a threat to validity concerning the similarity or difference of the perceived value of reuse. This was mitigated by using a common scoring system, resulting in ordinal rather than ratio metrics.

Generalizability "refers to the extent to which one can extend the account of a particular situation or population to other persons, times, or settings than those directly studied [90]." Here, the threat to validity consists of the fact that the research was performed within the confines of one corporation. While the corporation consists of several companies and cultures, there may also be an influence of the corporation itself. There also could be proprietary information that was not discussed. In addition, there may be a tendency to report successes and to keep working on or terminate efforts that have failed, success may be overreported. On the other hand, there might also be a tendency to report failures and the failures may be overreported. There may be an underreporting of results that were neither great successes nor failures. However, the results we obtained do not seem to indicate either situation. Once again, this study focused on large projects in the aerospace domain, and may not reflect smaller projects or other domains.

Evaluative Validity refers to the evaluative framework in deciding whether the reuse was, in fact, successful or not, and if so how much. The frameworks were likely to have differed in the different projects because the contexts were different. This was mitigated by considering the subjects' report of both reuse and the subjects' descriptive observations upon which the reports are based.

4.6 Conclusions and Future Work

This study investigated differences in approaches and developer experiences with various reuse strategies comparing embedded vs. nonembedded software systems. It considered reuse in development approaches such as ad hoc, model based, component based, product line, and COTS/GOTS reuse, as well as the artifacts that are reused. In a sense, this study is an update, since the oldest of these studies would not have been able to report on reuse in development approaches such as product line and component based software development ([42], [41]). We could determine that some results on reuse persist in the face of changes in system development technology. Results that persist include the preponderance of code

and requirements reuse and much less frequent architecture and design model reuse. There is extensive reporting of ad hoc reuse. Differences also presented themselves: reuse of models, test products and test clusters (the latter two especially in embedded systems) and the frequent reporting of product lines and model based development approaches coupled with reuse. As seen in the box plots, median outcomes did not differ between embedded and nonembedded systems but the variance was greater for embedded systems.

We began our analysis by recognizing an inherent bias in reuse approaches and reuse artifacts in terms of the frequency of their selection and how evolved the candidate artifacts and approaches are in systems with which we have experience. We further recognize a rank order exists among reusable artifacts and reuse approaches, again in terms of frequency and evolution. We formalize an identification of the relative ranking of these items which appear in our survey questions as multi-part responses. Through application of the Analytical Hierarchy Process, we derive the rank order of reuse components and reuse artifacts on the basis of frequency and evolution in a decision hierarchy. The order is expresses as numerical weights on a ratio scale. With these weights, we transform binary survey response data into ratio scale data upon which PCA operates.

Our selection of PCA in the analysis of the survey response data is motivated by our interest in relationships between survey responses which are not easily discovered through conventional analytical methods. Additionally, we are interested in comparing the outcome of MANOVA with that of PCA. Both MANOVA and PCA have their basis in statistical variance, so the analysis methods are mutually complimentary. PCA allows us to understand the survey data in terms of the relative variance in the responses to each survey question. These principal components express relationships between survey responses as discussed in the analysis of our results. Our strategy of selecting combinations of response categories which differ by one category allowed us to discover survey response relationships in the presence and absence of the variable category.

In general, our PCA confirmed that the selection of embedded vs nonembedded system type does not always relate to outcomes such as an increase or reduction in risk, or an

increase or reduction in test time. Some test pairs showed an independence in the relationship between reuse approaches and reuse artifacts, from system type. We draw conclusions cautiously from these results while recognizing opportunities for future research in which a greater volume of survey data is sought for a replicated study.

Some of the research questions were answered. For example, reuse effectiveness did not vary with project type (RQ 3), however, nuances of the development approaches did vary (RQ1). There was a significant difference in what artifacts were reused (RQ2). There is no statistical difference in reuse success based on the reporting of the outcome variables. Embedded systems were significantly more likely to reuse hardware, test products and test clusters. However, the reasons for this were not clear. However, respondents' comments imply that the tight coupling between hardware and software and the complexity of the performance requirements leads to a desire to reuse hardware/software components and their test suites once they have been proven.

Chapter 5

Semistructured Interview

The survey left us with more questions. We used a semistructured interview to see if those questions could be answered. In this chapter, we present the results of a set of semistructured interviews of experts in aerospace companies. These experts were asked about reuse practices, successes and failures, and the reasons why these happened. We were particularly interested in learning about differences and similarities in reuse approaches for embedded vs. nonembedded systems. In addition, since modern development approaches enable reuse of a wide variety of artifacts, we wanted to know whether artifact reuse was different between embedded and nonembedded systems and whether the experts thought certain development strategies worked better for one type of system than another, and why. Experts were from a variety of corporate cultures. Results indicate that there are important differences. For example, unlike nonembedded systems experts, embedded systems experts preferred platform standardization over platform independence. This is because of the need to optimize to a platform, important to embedded systems, whereas most nonembedded systems run in virtual environments and need to be platform neutral. Embedded systems experts prefer to use code already developed for the platform intact due to the difficulty of modifying optimized code.

5.1 Semistructured Interview Study Design

The results of this study, expected to be descriptive, are intended to add to a foundation to The results of this study, expected to be descriptive, are intended to explain reasons for quantitative results found in earlier study. Therefore, this study is designed to provide indicators of the direction needed for future research [5–7] .

Having learned from our previous research about the apparent different reuse outcomes between embedded and nonembedded systems in terms of development approaches and reused artifacts, we wanted to uncover the reasons for these differences. If we knew why these differences exist, it might be possible to identify practices that would improve reuse outcomes for both types of systems. To identify reasons for these differences, we decided to conduct semi-structured interviews of experts in both embedded and nonembedded systems. According to Rand [61], interviews "can be used as a primary data gathering method to collect information from individuals about their own practices, beliefs, or opinions. They can be used to gather information on past or present behaviors or experiences." These beliefs or opinions would be guided by their own experiences, both good and bad, with reusing software products. Rand [61] goes on to say that interviews "can further be used to gather background information or to tap into the expert knowledge of an individual. These interviews will likely gather factual material and data, such as descriptions of processes." From these experts, we can understand what is actually happening in industry, both in terms of what they are doing and what the outcomes are. Finally, Rand [61] points out that "semi-structured interviews are often used when the researcher wants to delve deeply into a topic and to understand thoroughly the answers provided." Since our goal is to understand the differences between embedded and nonembedded systems in reuse experience, as well as to understand differences in success using different development methods and reusing different artifacts, the semi-structured interview is the tool of choice.

According to Rand [61]. the process is:

- Frame the research
- Determine the sampling
- Design the questions
- Develop the protocol
- Prepare for the interview
- Conduct the interview
- Capture the data

The process we used for each of these steps is discussed below.

5.1.1 Frame the research

The context of this study is software engineering experiences within a corporation with many companies in the Aerospace industry. The study aims to understand software reuse practices and results in embedded and nonembedded systems and reasons for failure and success.

Theoretical Frame of Reference

There is a general belief that reuse is beneficial, in fact, it is required in many Government Requests for Proposal (RFP). In 1992, the Department of Defense published a Software Reuse Initiative requiring reuse. There is also a general assumption that reuse results in non-embedded systems are extensible to non-embedded systems [65].

Our theoretical framework is based on the two studies we conducted [5–7], the review of literature and the survey. We theorize that there are differences in reuse practices between embedded and nonembedded systems, and that while these lead to different reuse strategies and reused artifacts, both have similar outcomes.

As a result of this gap, we performed a survey of reuse practices across a large corporation, focusing on similarities and differences between embedded and nonembedded systems [6]. We asked about reuse success in each type of system to see if reuse worked comparably. We also asked about reuse strategies and artifacts to determine similarities and differences of reuse practices in embedded and nonembedded systems. This survey did find that, while reuse success between the types of systems were comparable, there was a significant difference both in the reuse strategies used and the artifacts reused. However, we were not able to determine exactly why this was the case.

Thus our theoretical framework is based on the two studies we conducted, the review of literature and the survey. We theorize that there are differences in reuse practices between embedded and nonembedded systems, and that while these lead to different reuse strategies and reused artifacts, both have similar outcomes.

Research Questions

Our first step was to identify the main research questions, what we hope to learn from the research. These are distinct from the questions we ask in the interviews. From the above theoretical framework and the results of our previous studies, we were left with the following questions:

- RQ-1 Is reuse really beneficial?
- RQ-2 Is reuse equally beneficial in embedded systems as non-embedded systems
- RQ-3 Do embedded systems use the same approaches as non-embedded systems
- RQ-4 Should embedded systems use the same approaches as non-embedded systems
- RQ-5 Does reuse effectiveness vary with project type (specifically embedded vs non-embedded systems)?
- RQ-6 Which reuse strategies are used?

- RQ-7 Which reuse artifacts are used?
- RQ-8 What is the reason for the difference in reuse strategies and reuse artifacts?

While some of these questions were answered by Andrews et. al. [6], we wanted to know why.

5.1.2 Sampling

There are nine different methods used to create a sample of subjects: Random, Systematic, Stratified, Structured, Cluster, Judgment, Convenience, Opportunity, and Snowball [61]. We selected the judgment method because "judgment sampling reflects some knowledge of the topic, so that people whose opinion will be important to the research, because of what you already know about them, will be selected [61]." While this method offers little ability to generalize, it does provide the best insight into reasons for outcomes in each of the areas of our research. Our goal is to understand reasons, not necessarily to generalize, which we have already done.

Candidate Selection

Once it was determined that the best method was judgment, we needed to develop a process for selecting the candidate subjects. To identify the best candidates, we obtained a list of fellows and distinguished technical staff from five companies, along with their areas of expertise and specific specialties. This list consisted of 134 individuals. From this list, we selected those individuals (54) who were either software or software systems experts.

We grouped the remaining 54 experts into three categories based on the company they worked for and their particular expertise. Group 1 consisted of the individuals most likely to have the most reuse outcome experience, balanced by the company type and type of system (embedded vs. nonembedded). 21 individuals were in this group. Group 2 was individuals less likely to have reuse outcome experience but still have significant important knowledge, or from companies with extensive reuse expertise. Group 2 consisted of 12 experts. Group

3 was those from whom we would expect less information or from companies with many experts. Group 3 consisted of 21 candidates. The grouping process involved the researcher and two colleagues, one from each of two companies, who knew all of the candidates.

Securing the Interviews

Once the candidates were identified, a letter was prepared to introduce the research and request the candidates' participation. The colleagues created cover letters requesting that the candidates agree to be interviewed. The letter of introduction and the questions were sent to the first 21 individuals, along with the letter from the researcher shown in Appendix A. From this correspondence, one candidate subject declined, six did not respond and 14 were willing to participate.

Confirming Sample Size

The next step was to determine sample size. We use the concept of data saturation discussed in Francis et. al. [46] to determine our sample size. Data saturation occurs at the point of data collection when no new additional information is found in relation to the research [51]. This is the point where, for the purpose of the study at hand, the appropriate sample size is a function of the range and distribution of the experiences rather than a need for statistical evaluation. In this study, while it is important to have an adequate sample size to explore the topic, the time of the candidate subjects is valuable and not to be taken lightly.

Francis et. al [46] recommends determining a number of interviews for a first round of analysis as an initial analysis sample. In their first example, the number was 10. We decided to use the 14 willing subjects as our initial sample. At this point we needed to determine a stopping criterion. Our approach was that after 14 interviews, if there had been no new information in answers to the main questions in the questionnaire for two consecutive interviews, we would conclude that data saturation had been reached. If, after the first 14 interviews data saturation had not been reached, the remainder of group 1

would be contacted again to see if they were willing to participate, then group 2, and finally, if necessary, group 3. As it turned out, data saturation was reached within the first 14 interviews.

5.1.3 Designing the questions

The questions were designed for the interviews to be open, in the sense that they allow the subject to privide his/her own answer, rather than closed, where the subject would select from a number of preselected answers. The open question method was selected because it does not limit the scope of the answer and thus offers the subject the opportunity to explain the phenomena s/he is reporting.

Developing the questions for our semistructured interviews required us to offer topics and questions to the subjects. These questions were carefully designed to elicit the subjects' ideas and opinions about their experiences with reuse, as opposed to leading the interviewee toward preconceived choices. The questions needed to allow the interviewer to follow predetermined questions with probes to get in-depth information on the topics. The two underlying principles were (1) to avoid leading the interview or imposing meanings, and (2) to create relaxed, comfortable conversation. In the process of developing the questions, the following elements were considered:

- Carefully plan the interview, with questions to ask and various ways of arranging them.
- Provide an overview of the purpose, your intended uses for the interview data, and the measures taken to protect confidentiality and anonymity. Discuss and get permission for tape recording.
- Ask a few background questions first, such as the subject's job title and responsibilities, time with the organisation, time in the industry, time working with reused products. These often provide necessary information create a comfortable interviewing environment.

- The questions should be broad, open-ended questions allowing the subject latitude in constructing an answer.
- Prepare, and save until later in the interview, questions on specific facts or other items of interest.
- Follow up questions should ask about statements without leading in the questioning.
- Use probes carefully to get more in-depth answers or to follow up on points of interest.

Multiplicity (triangulation) is obtained via selection of interviewees in different companies in different parts of the country working on a wide variety of systems. Preparation for the interviews consisted of

1. Craft questions in advance. We drafted an initial set of questions derived from the results of the questionnaire [8] using the above guidelines. The set of questions was presented to a focus group consisting of computer science doctoral candidates. The focus group recommended some refinement, which was implemented.
2. Conduct practice interview. This interview was conducted with a professional subject matter expert. Any confusion with the questions was noted, as well as any questions the interviewee felt were not asked. The practice subject and the interviewer discussed that it would be difficult to ask about all aspects of reuse the subjects might want to discuss.
3. Refine question. Based on the responses during the practice interview, it seemed some of the questions needed further explanation. One question was added, where the subject could make statements that the subject felt were important but not covered in the other questions. This was used to answer the problem that surfaced during the practice interview.
4. Prepare subjects with description of interview and consent forms. We emailed a letter of introduction to the subjects including a description of the interview and its

purpose, the list of questions, an explanation of the consent form and the consent form itself.

5.1.4 Developing the Protocol, Conducting the Interview and Data Collection

Our next step was to develop the protocol for conducting the interviews. A consistent protocol is important for reliable results. The protocol consisted of five segments: pre-interview, introduction, questions, wrap up and follow up.

In the pre-interview segment, the interviewer thanked the subject for his/her willingness to participate and arranged a mutually convenient one hour session over the telephone. The subject was reminded that the interview would be recorded and that s/he had been provided with the interview questions and the consent form. A calendar appointment confirmed the time and subject's phone number.

Once the appointment began and the subject contacted, the interviewer again thanked the subject for participating and again stated that the interview was being conducted over a speaker phone so it could be recorded and verified permission to record. The interviewer also asked if the subject had had an opportunity to review the questions and felt comfortable with them, stated that if the subject was uncomfortable with any question s/he was free to pass on the question, and restated that proprietary information whould not be discussed. The interviewer confirmed that the consent form was in her possession. The subject was encouraged to expand on his/her answers with opinions and anecdotes. The interviewer also informed the subject that once the interview was transcribed, it would be returned to the subject for review and corrections, and that if the subject had corrections or changes s/he could make them to the transcript and return to the interviewer. The returned transcript would become the transcript of record. Then the recorder was turned on. The interviewer stated the subject's name and asked for background information on the subject, such as years in software engineering, experience with reuse, etc.

The questions phase consisted of asking the questions in the survey, discussing the answers with probing questions and anecdotes. The final question was whether there was any additional information about reuse experience that had not been addressed by the questions in the survey.

The wrap-up phase consisted of the interviewer again thanking the subject for his/her time and thoughtful comments and a restatement that the transcript would be sent to the subject as soon as it was ready. The follow up segment consisted of transcribing the interview verbatim and sending the transcript to the subject.

5.1.5 Ethical Considerations

It is important to ensure the confidentiality of the interviews, both the records and the transcripts. In this study, the names of the interviewees were translated into a code, which is retained separate from the interviews. This was done prior to the data analysis, so there could be no leakage. The files are stored on a privately held USB flash drive that is not available to outsiders. Prior to the interview, the subjects were asked to sign consent forms detailing the nature of the research, the process and the promise of confidentiality. Each of the subjects signed the consent form, indicating faith in the ethics of the researcher.

The following ethical considerations were considered important and were discussed prior to the interviews:

- Do not give away proprietary information,
- Include consent form.
- Be sure subjects know they are being recorded.
- Remove names from answers (Person A, Person B, etc).
- Ensure conclusions about the company cannot be drawn from the information presented.
- Allow subjects to read transcripts for accuracy, and make any changes.

5.2 Results

Every subject responded with extensive anecdotal material and personal observations. Eight subjects returned transcripts, three had comments, and the other five were unchanged. The rest indicated that they accepted the transcripts as they were. The changes from the three with comments were made to the relevant transcripts, and these became the transcripts of record.

5.2.1 Summary of Responses

Question 1: What is your background in reuse?

Experience, both in terms of years and projects worked on, was of particular interest, in that it could offer an evolving picture of software development and reuse. The range of experience in embedded systems was from eight to 30 years, with an average of 22.3 years. For nonembedded systems it was 10 to 35 years with an average of 23.3 years. We concluded that the years of experience was comparable.

Question 2: What types of systems have you used reused products on - embedded vs nonembedded?

The responses were grouped into subjects who had worked primarily in embedded systems and those who had worked primarily in nonembedded systems. Seven subjects were primarily involved in embedded systems, six primarily nonembedded systems, one had worked in both embedded and nonembedded systems equally. The one who had worked in both embedded systems and nonembedded systems was especially interesting for the comparisons and was asked to identify whether his answer referred to embedded or nonembedded systems when there was a difference. These responses are included in both embedded and nonembedded system categories unless specified otherwise. Seven subjects were primarily involved in embedded systems, six primarily nonembedded systems, one worked on both.

Embedded systems experts interviewed worked on projects such as satellites, missiles, and aircraft. Specialties included Field Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), Guidance, Navigation and Control (GNC), Command and Data Handling (CDH), Communications, sensors, Power Distribution Units (PDUs) and Health and Status. Nonembedded systems experts worked on projects such as ground control systems, web-based distribution systems, information systems, enterprise systems and big data. Applications included satellite and missile ground control, air traffic control, postal services, health information services, logistics, data processing, and information distribution services.

Question 3: What reuse strategies have you used?

Reuse strategies in this case referred to development approach. The responses to the development approach were somewhat more varied. In embedded systems, four used ad hoc reuse, three used model based development, one used component based software development, three used product line and one used ontology. The ad hoc reuse was explained by embedded systems experts as being the result of having developed similar systems repeatedly, and reusing these software products was the normal approach. While many embedded systems experts cited model based development as one of their reuse strategies, their definition of model based development included the use of any models, and the models they were reusing were performance models and simulations. One used what he called mega reuse, that is, he took the software intact and did not modify it, and one described his development approach as a framework. Some subjects used multiple approaches. Three subjects used ad hoc reuse and model based development, two used ad hoc reuse and product line development, one used model based development and ontology. In nonembedded systems, four used ad hoc reuse, one used component based software development, one used model based development, one used the mega reuse discussed above, one used a framework and one described his development approach as SOA (a type of model based developement). Nonembedded systems experts explained their use of ad hoc development in that many of

the capabilities they needed were readily available in either freeware websites or available from prior projects. The one using component based software development had inherited the architecture and components and was upgrading an existing system. Again, some subjects used multiple approaches. One used ad hoc reuse and model based development and one used ad hoc reuse and model based development. Much of the ad hoc reuse can be attributed to the fact that many of these projects have been ongoing for a long time.

Table 5.1 shows the frequency of development approaches used by these software experts. The first column identifies the development approach, the second shows the number of embedded systems experts who identified that development approach as part of their development strategy, the third column is the per cent of the embedded systems experts using that development approach (number of embedded systems experts citing use of the approach divided by total number of embedded systems experts), the fourth column shows the number of nonembedded systems experts using that approach, the fifth column shows the per cent of nonembedded experts using the approach, and the final column is the difference in the per cent of embedded systems experts using the approach and the nonembedded systems experts using that approach. The same analysis was performed on Tables 5.2, 5.4 and 5.6. The reasons the experts gave for their use of development approaches are discussed in subsection 5.3.1.

Table 5.1: Responses to Development Approach Used

Approach	**Embedded**		**Nonembedded**		**Total**	
Ad Hoc	4	50%	4	57%	8	53%
Component	1	13%	1	14%	2	13%
Model	3	38%	1	14%	4	27%
Product Line	3	38%	0	0%	3	20%
Ontology	1	13%	0	0%	1	7%
Mega	1	13%	1	14%	2	13%
Framework	1	13%	1	14%	2	13%
SOA	0	0%	1	14%	1	7%

* Note: Subject who worked both embedded and nonembedded systems counted twice

Question 4 What artifacts have you reused?

Table 5.2: Responses to Artifacts Used

Artifact	Embedded		Nonembedded		Total	
Algorithms	1	13%	1	14%	2	13%
Architecture	2	25%	5	71%	7	47%
ArchitectureModels	2	25%	1	14%	3	20%
Code	8	100%	7	100%	15	100%
Components	5	63%	0	0%	5	33%
COTS	3	38%	2	29%	5	33%
Data,database	1	13%	2	29%	3	20%
Design	8	100%	7	100%	15	100%
Design Models	5	63%	5	71%	10	67%
Design Patterns	1	13%	2	29%	3	20%
Design Products	0	0%	5	71%	5	33%
Documentation	4	50%	2	29%	6	40%
Hardware	4	50%	3	43%	7	47%
Interfaces	3	38%	3	43%	6	40%
Models	2	25%	0	0%	2	13%
Performance Models	4	50%	2	29%	6	40%
Requirements	8	100%	7	100%	15	100%
Services	0	0%	1	14%	1	7%
Simulations	3	38%	2	29%	5	33%
Service Level Agreements	0	0%	3	43%	3	20%
Test Clusters	4	50%	2	29%	6	40%
Test Environment	0	0%	1	14%	1	7%
Test Products	6	75%	4	57%	10	67%
Tool Suite	0	0%	1	14%	1	7%
Use Cases	2	25%	1	14%	1	7%

* Note: Subject who worked both embedded and nonembedded systems counted twice

The responses to the question about artifacts used is shown in Table 5.2. As we can see from the responses, every subject reused code, requirements and design. Several experts explained that these were reused because they could be used intact, and had already been proven on previous projects. However, we begin to see differences between embedded and nonembedded systems in other artifacts. For example, nonembedded systems experts were far more likely to report reusing architecture (71%) than embedded systems experts (25%). The architectures were easy to modify and could import exactly the services they intended to use. 63% of embedded systems experts reported reusing components, compared to zero for nonembedded systems. Embedded systems experts explained that reusing existing

components provided known performance and reduced risk. No embedded systems experts reported use of SOA, but 43% of nonembedded systems experts did. Test Products reported use by 75% of embedded systems experts, compared with only 57% of nonembedded systems experts. Embedded systems experts explained that the test products, such as test drivers and test data had been developed specifically for the components they were reusing and tested the reused requirements. No reuse of design products was reported by embedded systems experts, while 71% of nonembedded systems experts did use them. Embedded systems experts stated that in many cases, the designs had been used for so long, the design products did not exist. The rest of the artifacts reported similar levels of reuse.

Question 5 What was the level of reuse?

The level of reuse is measured in percentages, and ranges from 5% to 90% in embedded, 15% to 90% in nonembedded systems. Table 5.3 shows the levels of reuse reported. In cases where the reuse level was 80-90%, the reuse was a modification to an existing system. In these cases, the reported reuse was the portion of the system that was not changed in the modification, but a new contractor would have to develop if they were awarded the contract.

Table 5.3: Responses to Level of Reuse

%	Embedded	Nonembedded
80-90	3	4
70	0	1
50	2	1
30	1	1
20	3	1
15	0	1
5	1	0

* Note: Note that some experts reported reuse levels on more than one project

Question 6 Have you encountered any obstacles?

Most commonly mentioned obstacles to successful reuse are shown in Table 5.4. Selected responses to obstacles to reuse are shown in Table 5.5. Please note that some respondents had more than one comment for certain obstacles.

We see that every expert interviewed cited modification and understanding as major obstacles, difficulty, whether perceived or real was cited by all but 2, fit was cited by 11, platform dependence cited by 10, and the loss of flexibility or freedom, obsolescence and culture cited by nine. All experts in embedded systems found "fit" to be an obstacle, as did four in nonembedded systems. Platform dependence was also a major obstacle.

Commitment was the obstacle cited most by embedded systems experts compared with nonembedded systems experts (46% difference). Existing culture was next, with a difference of 34%, then culture at 32%, fit at 30%, lack of trust of the inherited products was cited by 25% more embedded systems experts than by nonembedded systems experts. On the other end of the scale, the certification process was cited 45% more by nonembedded systems experts than embedded systems experts, followed by maintenance (30%), the risk of inserting new defects (29%), and difficulties, real or perceived (25%). Out of 24 obstacles cited, eight were cited at least 20% more often by embedded systems experts than nonembedded systems experts, and seven were cited at least 20% more often by nonembedded systems experts than embedded systems experts. Only nine were within 10% difference. Clearly, the greatest obstacles to reuse are different between embedded systems and nonembedded systems.

It is interesting to observe the obstacles and their relationship to various artifacts. For this analysis, we remove commitment, culture, lack of trust, lack of metrics, contracts and individualism, as comments about those obstacles did not address artifacts or development method. We look at existing defects, fit, easier to build from scratch, obsolescence/age, modification, understanding, complexity, lack of documentation, forking, platform dependence, performance goals, difficulty, insert defects, maintenance and certification process

Table 5.4: Responses to Obstacles Encountered

Obstacles/Type	**Embedded**		**Nonembedded**		**Total**		**Delta**
Commitment	6	75%	2	29%	8	53%	46%
Existing Defects	5	63%	2	29%	7	47%	34%
Culture	6	75%	3	43%	9	60%	32%
Fit	7	88%	4	57%	11	73%	30%
Lack of trust	2	25%	0	0%	2	13%	25%
Lack of Metrics	5	63%	3	43%	8	53%	20%
Contracts	5	63%	3	43%	8	53%	20%
Individualism	5	63%	3	43%	8	53%	20%
Easier to build from scratch	4	50%	3	43%	7	47%	7%
Obsolescence, age	5	63%	4	57%	9	60%	5%
Loss of flexibility or freedom	5	63%	4	57%	9	60%	5%
Modification	8	100%	7	100%	14	93%	0%
Understanding	8	100%	7	100%	14	93%	0%
Complexity	2	25%	2	29%	4	27%	-4%
Lack of Documentation	4	50%	4	57%	8	53%	-7%
Forking	4	50%	4	57%	8	53%	-7%
Platform Dependence	5	63%	5	71%	10	67%	-9%
Unintended Consequences	3	38%	4	57%	7	47%	-20%
Missed Opportunities	3	38%	4	57%	7	47%	-20%
Performance Goals	3	38%	4	57%	7	47%	-20%
Difficulty - real or perceived	6	75%	7	100%	13	87%	-25%
Insert defects	0	0%	2	29%	2	13%	-29%
Maintenance	1	13%	3	43%	4	27%	-30%
Certification Process	1	13%	4	57%	5	33%	-45%

* Note: Subject who worked both embedded and nonembedded systems counted twice

to see which of these obstacles are associated with certain artifacts. We find that artifacts associated with:

- Existing Defects mentions problems with code three times, and no other artifacts are mentioned. The experts indicated that has already been deployed is expected to be defect free, but may not have been tested for a circumstance that is exposed by the new system. Once a test indicates a defect in the code, the defect is often be difficult to find and correct.

- Fit mentions problems with code three times, requirements two times, components once and design once. Sometimes, the documentation on code selected for reuse indicates that it satisfies the needs of the project, however, once the project attempts to use that code, it does not integrate into the rest of the code (because of incompatible interfaces, coding conventions, approach). Sometimes the requirements being reused are different from the goals of the customer. Sometimes the design approach is different from the approach taken by the reusing project.

- Easier to build from scratch mentions problems with code five times, and no other artifacts are mentioned. Each expert who mentioned this obstacle indicated that when the code needs modification, the time and effort needed to identify the changes, make the changes, and then trace their impact on the rest of the code is greater than would be needed to write new code.

- Obsolescence/age mentions problems with hardware four times and code three times. Experts mentioned that often the hardware the code was developed for is no longer available, and that adapting code for new hardware was more difficult than writing new code. They mentioned that often code is written in an obsolete language.

- Loss of flexibility or freedom mentions components once. When a component is controlled and has to be used intact, that dictates that portion of the solution.

- Modification mentions problems with code 14 times, requirements twice, and components, design and architecture each once. Every system expert indicated that once code needs to be modified, unexpected problems surface.

- Understanding mentions problems with code four times and documentation once. Several experts stated that it is difficult to understand code written by another developer, especially when that developer was writing to a different set of coding practices.

- Complexity mentions problems with code four times, components two times, and COTS and requirements once each. The experts discussed the difficulty of following threads through complex code to know exactly what was happening.

- Lack of documentation mentions problems with documentation eight times, code and requirements twice each, and design once.

- Forking mentions problems with code nine times and software components once. The experts explain this as creating several copies of the same artifact with minor variations, which requires separate control and maintenance, negating the benefits of reuse.

- Platform dependence mentions problems with code 12 times, hardware eleven times and design once. Experts cite the difficulties of reusing code when the platform changes.

- Performance goals mention code five times, requirements three times, components once and design once. Experts discuss the idea that often these performance requirements are hard coded and difficult to modify for a different set of performance goals.

- Difficulty mentions code three times and components twice.

- Certification mentions code twice. According to the experts, the certification standards differ across projects, and the code needs to be modified to accommodate the differences in these standards.

We did not find any mention of development methods mentioned in the statements about obstacles. It is interesting how many of these obstacles, other than documentation, appear to be based on reuse of code more often than any other artifact.

Table 5.5: Expert Comments on Obstacles.

Obstacle	Select Expert Comments
Commitment	It's pushed to the back burner and not accepted by the program. Until we do that on a big scale, multiple programs, no one will listen. Nobody in the business areas felt ownership. It's too easy not to bother. You've got to be committed to a certain mindset to try to find it and see if you can make it work.
Existing Defects	There could be defects that were undiscovered from the previous use because they never went through that condition. Some of the problems you might be inheriting as well. You run into problems that were in the code. You may be inheriting software that doesn't have the quality you need. You inherit software irregularities or inconsistencies, if it's got defects.
Culture	Anything that is successful develops either a set of supporters or set of opponents. It's a not invented here kind of attitude. They don't care about the next program down the pipe. The biggest obstacle to software reuse is more cultural and people driven, not technical driven. There is a lack of collaboration.
Fit	We didn't understand the context of the reuse, or what we were going to have to develop and underestimated the amount of change necessary to make the reuse useful in the new program. When you're matching a component to your needs, you often exceed some design parameter. You assume reusability without checking to see if there's a match. It's not always a clean fit for the application that you're working on. The existing requirements aren't appropriate. The people who chose the reuse didn't dig deep enough to understand how much the design and the requirements, and then the code, would have to change. You're fitting existing software into a system that's not a perfect fit.
Lack of Trust	Finding things wasn't the biggest challenge, it was deciding whether they could be trusted. They wouldn't use something they found unless they could satisfy themselves about its pedigree. Developers don't trust work other people have done, especially it was developed for a different purpose.
Lack of Metrics	You have to know how long it takes, across the life cycle. You have to know how bad you're doing to be able to do things differently. We don't have metrics. I don't know if anybody has collected any metrics, I don't know of any available out there. We rely too heavily on ELOC or SLOC counts. I'm not sure SLOC counts make sense as a basis for cost estimation.
Contracts	The way the contract is written doesn't really enable reuse. If customers don't dictate reuse they won't pay for developing things for reuse. Customers put clauses and constraints on the reuse between their programs and other programs. They often require customer approval. That kills a lot of reuse from the gate.
Individualism "Not Invented Here"	They think they can do a better job even though they know that they're going to spend a similar amount of time doing it. "Not invented here" is often the most significant barrier. Engineers like to invent stuff and reuse means that you get someone else's hand me downs, instead of something new. When it becomes custom everyone's got a different way of doing something, and their way is always the best.

Continued on next page

Table 5.5 – continued from previous page

Obstacle	Select Respondent Comments
Easier to Build from Scratch	If it costs you more than 30%, you might as well rewrite it. In some cases it took longer to reuse code because of all the issues with it, then it would have been to just do it from scratch. When you are modifying about 2/3 of it, you,Äôre losing money, you would have been better off to start from scratch. Why would a software developer spend 10 minutes looking for something he could easily write in a day or two? It's an inferior solution to what it would have been if you had just done it from scratch. It's less risk if I schedule to build it myself.
Obsolescence, age	The people that were the go to people before aren't producing the same parts anymore. Can you imagine trying to reuse 30-40 year old assembly code? What can be a problem is if the sensor technology changes radically, then the existing requirements are not appropriate. Often I've got some obsolete stuff, I've got to move stuff from one platform to another platform, or from one language which we consider obsolete, to another language. We are bridging technologies across a decade or more and the technology in hardware and software is evolving so rapidly that who would want to reuse something that's 10 years old?
Loss of Flexibility or Freedom	The more people that you have reusing a component the less freedom of movement you have for any individual stakeholder in that reuse. When you share something you don't get 100% say in what that something is any more. Reuse constrains innovation to a degree. It won't give the best performance or the most eloquent design for that particular situation. The shortcomings of reuse is that often you're taking a sub optimal design for the problem at hand.
Modification	If it's very dissimilar you will probably have to make so many modifications that it's not worthwhile. If the reuse is doesn't meet performance requirements, you have to evaluate what you have to do to that reused software. If I have to redesign and rebuild it it may not be worth the reuse. When you try to match a component to the needs, and exceed some design parameter, you need to determine whether you can adapt or adopt it. When they tried to do a slight modification, the whole thing fell down, it was all patched together, and we had to well throw it away and start over. If you want to modify it, and the code is not written in a readable, modifiable manner you get into all kinds of trouble. The more you modify, the more chance that you're going to lose money. There were unique requirements for our program that required changes we didn't anticipate. Code reuse is almost always a recipe for failure. I get into the code and decide to use this and not that, and modify this, picking code apart at that level of granularity, I've never seen it succeed. In something as simple as the way you name variables involves rewrite of all the variables and how you do entry points, and how you send commands and collect telemetry, it ends up being not reusable, because of the significant modification necessary to go be compatible with the software architecture.

Continued on next page

Table 5.5 – continued from previous page

Obstacle	Select Respondent Comments
Understanding	Either we didn't understand the exact context of the reuse, we didn't understand what we were getting or we didn't accurately understand what we were going to have to develop and underestimated the amount of change that would become necessary to make the reuse actually useful in the new program. How do I interface it? Part of the up front cost is learning what the other guy did. Sometimes they've named things in ways that aren't immediately obvious. If you cram something in there that isn't a right fit because you didn't understand what the reuse did, or what the system you want to put it in had to do, you're going to fail.
Complexity	It has all these various COTS packages. Each line of code takes a longer to write, but one line of code contains so much more complexity raw software that's coded it's very difficult to do analysis. People get a pile of code they have to slog through and develop their own mental image or mental model of it. As the lowest level of detail components get aggregated together and as those aggregations get larger and larger the ability to juggle those individual components mentally requires some abstraction, so you need models to be able to make sense of it. The requirements base can be fairly large.
Lack of Documentation	They found out that the requirements document didn,Äôt match what the design was, and the design description didn,Äôt match either one. What would make it easier is higher quality documentation for the as built systems. We need a reusable description of what the software does, so that whoever reuses that service has an idea of exactly what they're getting rather than having to just dive into the code to figure it out. It requires a lot of thought and documentation. Is the documentation adequate, do we understand it enough that we can modify it? We find poor requirements, lack of documentation, nothing seems to work correctly.
Forking	That new system takes over, modifying it into a variant of that baseline and duplicated the total life cycle cost of maintaining what becomes unique software. Failure is if you fork the baseline and then run with a different version. Now you have 2 copies of the code. You're trying to maintain one managed control code set, and they copy it and go off and fork it. You never know what they do. People will run into problems in the code, but since they've forked, they don't have the benefit of getting the fixes for the problems that were in the code, they don't get advances when you update the design and add more features to the design and implementation, they lose those as well. I want a consistent, stable, core group of software components.

Continued on next page

Table 5.5 – continued from previous page

Obstacle	Select Respondent Comments
Platform Dependence	You have software you know runs on a certain server and operating system, you,Äôre confined to using that. The machine architecture you,Äôre using is going to influence what is and isn't reusable. Low level coding is a misnomer when you say portable, given the complexity of the current generations of hardware. If the next embedded system had a different processor, we may not have used a lot in the way of designs. It may not have the same computer language and differences with compilers, development environments, means you had to mess with the code. If you change the underlying hardware including the CPU, processor, and hardware that interfaces to the digital hardware, you might have to redo 80% of your code even if the functionality is identical. If you change the hardware, typically different hardware has different control and status interfaces and messages and response requirements. We have a lot of flux in the processor world. Anytime we're changing hardware we are also changing operating systems. For scientific algorithms if we move from one type of the processor to another, all the underlying mathematical libraries changed. Platform independence causes a problem.
Unintended Consequences	Our customer is coming to grips with some of the collateral aspects of unintended consequences of reuse. Both the company and the customer are discovering that that,Äôs the unintended consequences.
Missed Opportunities	It's more common not to even try to reuse. It's more often we discover missed opportunities for reuse, where we didn't know that somebody on another program had built something similar that we could have reused. We need access to the information to know what there is to reuse and where it is, what requirements it matches, and the design.
Performance Goals	We defined timelines based on the requirements, and how the component supports that performance. If the reuse doesn't meet your performance requirements, then you have to evaluate what you have to do to that reused software so that it will meet performance requirements, and if I have to redesign it then it may not be worth the reuse. Getting that algorithm to meet timelines and work repeatedly in an operational sense is a big deal. When we have to do the science to ops transformations, we've been off by 100%. Performance aspects increase risk.
Difficulty - real or perceived	There is the feeling that it's going to be more effort than to just start out from scratch. There are borderline cases, where you say let's just try to reuse it just make a few changes that end up being terrible experiences. You are wasting your time and its more frustration than it's worth. It never goes quite as smoothly as we thought it would. When people build software components, it becomes very difficult to reuse. In something as simple as the way you name the variables. I don't know it, I don't understand it, I would have to make too many changes, it would take me too long to understand what it did.
Insert Defects	You can introduce problems. You introduce errors in the product that you didn't expect. We rarely understand how it's going to ripple through the software and impact all the COTS packages.

Continued on next page

Table 5.5 – continued from previous page

Obstacle	Select Respondent Comments
Maintenance	We duplicated the total life cycle cost of maintaining what now becomes somewhat unique software. While you did reduce some of your up front development cost, you negated most of the savings on the back side of the life cycle. For systems that live more than a couple of years, the Operations and Maintenance tail is the expensive piece.
Certification Process	You have to go through the certification process if you haven't done it. As we move into the new standard environment, less of what we have can be reused and the cost is significant. Depending on where that software is going to operate, it can restrict you from any reuse.

Question 7 How do you define success/failure?

Almost all of the respondents shared a definition of success as improvement in cost and schedule, both in embedded and nonembedded systems. Similarly, there was agreement that the ability to estimate cost and schedule savings were part of the success in reuse.

Question 8 How successful were you?

When asked how successful they were, experts working on nonembedded systems reported slightly better results, with one reporting consistent success, two reporting being mostly successful, one reporting "hit and miss" and one reporting mostly not. One expert working on embedded systems reported mostly successful, two reported "hit and miss," one reported missing estimates 35% of the time and two reported that they were unable to tell because they had no metrics.

Question 9 What drove the success or failure?

Summaries of the reasons for success of reuse are reported in Table 5.6. (The failure factors tended to reflect the obstacles listed above.) However, in the coding process, we were able to identify several other factors for success based on statements made in answer to other questions.

Table 5.6: Success Factors for Softare Reuse

Success Factor	E	E %	N	N %	T	T %	Delta
Planning up front	4	50%	2	29%	6	40%	21%
Control (authoritarian)	2	25%	3	43%	5	33%	-18%
Design for Reuse	3	38%	1	14%	4	27%	23%
Documentation	3	38%	1	14%	4	27%	23%
Experience	2	25%	2	29%	4	27%	-4%
Similar Product or Environment	2	25%	1	14%	3	20%	11%
Consistent baseline	1	13%	1	14%	2	13%	-2%
Automated testing		0%	1	14%	1	7%	-14%
Culture		0%	1	14%	1	7%	-14%
Forecast future demand	1	13%		0%	1	7%	13%
Graphical Representation	1	13%		0%	1	7%	13%
Metrics	1	13%		0%	1	7%	13%
Product Line	1	13%		0%	1	7%	13%
Service Oriented Architecture		0%	1	14%	1	7%	-14%

* Note: Subject who worked both embedded and nonembedded systems counted twice

Both embedded systems experts and nonembedded systems experts reported planning up front, control, documentation and experience as keys to successful reuse. Table 5.7 provides insight into how respondents explained these success factors and how they added to reuse success. It is interesting to notice how many more comments were made about obstacles than success factors. It is also interesting that the development approach was mentioned as a success factor twice (product line and SOA), where it was not mentioned in the obstacles. It is also worth noting that automated tests and models (graphical representation) are the only artifacts mentioned in the discussion of success factors. Interestingly, code reuse was not mentioned as a success factor.

Table 5.7: Respondent Comments on Success Factors.

Success Factor	**Select Respondent Comments**
Planning Up Front	I did a survey of the things that cause the most issues for people. Up front you spend some time looking, evaluating the current design of what we are going to try to reuse. You have to do up front work. You spend a little up front to save a lot on the back end. What drives success is the degree to which we follow our guidance for evaluating the reuse that we are going to inherit.

Continued on next page

Table 5.7 – continued from previous page

Success Factor	Select Respondent Comments
Control (authoritarian)	It works well when they control it. When I do reuse, I am very authoritarian about how it's done. I am much more authoritarian about tracking reuse than new development. That's how you calculate your latencies and throughputs. You lose control of those when you lose control of the component.
Design for Reuse	Unless products are built for reuse, reuse becomes extremely problematic. If you don't plan and design for reuse you won't really be able to do it. We're doing better with trying to design for reuse.
Documentation	We have to have the documentation of the requirements for the software that you want to reuse. We need a reusable description of what the software does, so that whoever reuses it has an idea of what they're getting rather than having to dive into the code to figure it out. The key is to capture enough metadata about the reuse artifact that a person can determine in a reasonable amount of time whether that component is one you want to invest in. One of the problems we have with software reuse is that we haven't carefully documented after CDR the changes that occurred.
Experience	We had meeting after meeting with all the experts to make these decisions. The code that we reuse is internal, we have access to it, we know the people who wrote it, they can tell us the worms about it upfront so we can say at this is good with this is not good. We really need some expertise. We bring in what I would call algorithm-smart software engineers who understand the algorithm science. You hope that the person is a person who had worked on the stuff in the past and had a good understanding of what's going to need to be changed. If that isn't true, all bets are off. So either you built it and know how to use it or you practiced under someone who knows how to use it, or you've learned through tribal knowledge how to use it.
Similar Product or Environment	Success is driven by how similar the new environment is, how similar the use is. If it's very dissimilar you'd better be careful. Your software may function a certain way in your environment but when you move it to another environment there are environmental impacts and it may perform differently. I'm looking at similar code or artifacts in similar projects and adapting them for my use. I think that reuse and commonality go together. You get some inherent ability to reuse because of the commonality of the underlying system. If you come along with a system that has something different, then of course they don't have much in terms of reuse.
Consistent baseline	We maintain the discipline of having one baseline across multiple programs and maintaining that one baseline. I have a common layer of utilities that we all use. We make sure it's appropriately registered and baselined. When we reuse, it's not as though we're changing the development environment or the operational environment.

Continued on next page

Table 5.7 – continued from previous page

Success Factor	Select Respondent Comments
Automated testing	Automated tests are becoming more prolific. I have standardized design products and performance models and self testing software. What has really been liberating in the last few years for us has been automated testing. Automated testing has probably been the most significant piece. Auto testing, the result is automatically tested so you don't have to manually look at it but it can be run as a regression test.
Culture	The whole concept of reuse becomes something that's just a natural part of your design methodology. It's been a cultural shift for us as we try to position ourselves to align with the marketplace. You have to have the culture for it. You need the mindset and the commitment to do it. It may be more of a factor of the environment and the people and the tools than it is the actual amount of code they are reusing.
Forecast future demand	The biggest thing is the ability to correctly forecast future demand. If you're doing reuse in a product family, you need to forecast market demand for products in your product family.
Graphical Representation	One strength of the human species is the ability to grasp complex concepts via graphical representations like models. We inherited a software design that was graphical. We were able to quickly understand the capabilities of the reuse code, it was documented very well and graphical, and we understood what it did, and it actually executed.
Metrics	So the metrics are something we're pushing right now. Metrics is the basis for all process improvement.
Product Line	There's a big bang for the buck there if you're working product lines or a domain. If we've got something within a product line and we're creating new versions of the product line, very little is not reusable. You get benefits across the board with a product line
Service Oriented Architecture	Service Oriented Architecture is liberating in that the original component is designed to be a reusable service rather than a point solution for a particular application, that has been revolutionary. By moving to service oriented architectures we've had a greater emphasis on reusing COTS software. By having a service oriented architecture, we could modularize and define the interfaces, we could also make a nice interface between the workflow control and the scientific algorithms.

Question 10 How do you define component? Model?

The respondents had a variety of definitions for components and models. To describe componens, embedded systems developers used expressions like a physical entity (3), performs a discrete function (3), decoupled (2), interfaces (2) and domain (1). Three said it was software only, one said it was hardware and software together, three said it was

software during development and hardware and software together during integration and test. Thus, the last group could either be reusing a software component or a component comprised of both hardware and software. Three respondents did not like to use the word component because its meaning is too ambiguous. One said a component was whatever the customer said it is. Developers of nonembedded systems described components as interfaces (2), a discrete capability (2), inputs and outputs (1), representation of physical thing (1), cohesive, loosely coupled thing (1), service (1) and code plus documentation (1).

Models were also described in a variety of ways. Developers of embedded systems described models as a representation (2), and in particular a design representation (3), an architecture representation (3), or a graphical representation (1). Five said a model was a simulation. Four said it was a gauge of performance (4). One said it was a way to capture the essence of a system or software. One did not like the term. Developers of nonembedded systems used terms such as performance (2), design patterns (2), a representation (2), UML (1), development process (1), framework (1), simulation (1), and a representation of how components interact (1).

The experts' statements made it clear that the respondents were talking about different types of models. We separated the types of models into four categories, architecture models, design models, performance models and simulations. Table 5.8 shows the different models and the number of comments about those models made by the respondents.

Table 5.8: Type ∩ Rating ∩ Models

	Positive		Negative		Information	
Artifact	Embedded	Nonembedded	Embedded	Nonembedded	Embedded	Nonembedded
Models	70	35	9	6	13	13
Architecture Models	16	13	2	1	6	2
Design Models	21	15	2	3	9	3
Performance Models	13	14	1	1	4	8
Simulations	16	4	0	1	8	4

* Models listed in the order of phase.

The table shows us that the models reused by embedded systems experts are different from the models reused by nonembedded systems experts. Even the definitions of model based differ. When embedded systems experts say they are using a model based approach, often they mean that their approach uses models, referring to design models, performance models and simulations. Nonembedded systems experts often use a more rigorous definition of model based development. Person K said, "I think something that's model based, the model based engineering (MBE) approach, I think incorporates all those together in an integrated set of models. It would be your design model, your performance models, your simulation models, your architecture models, it's everything linked together." Person E said, "I think that a fully model-based development would include not only models of the system and software development but also models of the environment and aspects of the environment that are pertinent." Even their design models differ from the design models of nonembedded systems. Embedded systems are more likely to use Simulink as both a design modeling tool and a performance modeling tool. Person D said, "We know the tools that enable SySML and Simulink, where you develop a model of the device, either as you are trying to develop the lower level entities or you develop your algorithm application." They also generate their code from that model. Nonembedded systems are more likely to use a UML or SySML modeling tool like Rhapsody or Rational Rose for their design models, simultaneously creating integrated architecture models. So on the one extreme, model based development is a totally integrated set of models,with an entire suite of modeling artifacts reused. On the other extreme it is anything that uses models. These different definition of what artifacts imply model based development could cause the model based development reuse approach to either be overcounted or undercounted.

Question 11 What were the benefits of reuse? Non-benefits?

Benefits of reuse turned out to be very much like success factors. The ones most often mentioned by embedded systems developers were cost reduction(4), time or schedule reducrion (5), cleaner development (2), reduction in defects (2) and reduction in risk (2).

Developers of nonembedded systems mentioned cost reduction(4), time reduction (2), and reduced complexity (1). Nonbenefits of reuse were much the same as reasons for failure.

When asked about the benefits of reuse in terms of outcomes, every subject interviewed agreed that reuse reduced cost. However, every embedded systems expert and four nonembedded systems experts expressed concerns that reuse could also increase costs. In embedded systems, poor documentation, latent defects, needed modifications, platform dependence, different performance goals and different nonfunctional requirements were cited as reasons reusing existing code would cost more than rewriting it. Embedded systems experts cited maintenance, obsolescence, platform dependence, modification and recertification as reasons reusing could cost more. These explanations were often related to code reuse. Neither embedded nor nonembedded systems experts cited any negative impacts with reusing any other artifacts.

Four embedded systems experts said that reuse saves hours in development and test, as did three nonembedded systems experts. The time savings for both types of systems were attributed to having a similar environment, a consistent or controlled baseline, and processes and procedures. Three embedded systems experts found that reuse can wind up taking more time than developing the software. The additional time to reuse was the result of poor documentation and having to fix existing defects. Five embedded systems experts mentioned performance as a positive reason for reuse, as did three nonembedded systems experts. For embedded systems experts, the reuse of high performance algorithms was a major savings over trying to redevelop them. Nonembedded systems experts reused performance models. They asserted that it was easier to modify these models to meet performance requirements than to build new models. The one embedded systems expert who felt that reuse could hurt performance cited a lack of understanding of the code and poor fit into the new system.

Question 12 What about reuse and nonfunctional requirements?

Nonfunctional requirements offered the greatest variation of opinion, with many factors figuring into reusability as shown in Table 5.9. Overall, nonfunctional requirements interfered with successful reuse. Nonfunctional requirements were often different from project to project, requiring code modification as the reused code was ported. This was particularly true with performance requirements. Changing standards, particularly in security, meant that previously accepted software had to be reworked. Experts also cited the difficulty of selling off (obtaining customer aggreement that the requirements were met) the nonfunctional requirements of reused code. However, when the nonfunctional requirements were the same across the projects, reusing existing artifacts was useful.

Table 5.9: Reuse and Nonfunctional Requirements

	Embedded	Nonembedded
Reuse makes it easier	If standards are used in reused product Nonfunctional requirements that persist across programs Design patterns help reuse	If the reuse meets requirements Design patterns help reuse Nonfunctional requirements that persist across programs
Reuse makes it harder	Unique nonfunctional requirements interfere Hard to track to requirements Security requirements inhibit reuse Force changes in architecture and design Hard to recertify Quicker to build from scratch	Unique nonfunctional requirements interfere Hard capturing metadata about artifacts New security requirements make it hard Desirements Hard to sell off Product latency Changing standards Harder or impossible if requirements are different

Question 13 In your opinion does reusing the hardware make a difference? Why or why not?

All embedded system developers and all but one nonembedded system developer agreed that using the hardware makes a difference in software reuse success. However, they diverged on the question of platform independence. Three embedded systems developers said that platform independence was a detriment to successful reuse, two said it was not a goal. Instead, three favored platform standardization and two favored platform portability enabled by standardization. One prefered platform independent models that could be reused, and that had the capability to autogenerate the code. In nonembedded systems, four favored platform independence, and two favored platform standardization. One was concerned about porting costs.

Findings from the answers to the questions The most used artifacts regardless of system type were code and requirements, with 12 each, design models and test products with 10 each, architecture, designs, hardware with seven each, and test clusters and performance with six each. The rest had five or fewer responses. In embedded systems, all respondents used code and requirements, six used test products, five used components and design models the rest were cited by four or less. In nonembedded systems, architecture, code, design models, design products and requirements were used by five, test products were cited by four and the rest were cited by three or fewer respondents. Thus almost everybody is reusing code and requirements, but after that point they begin to diverge. Not one embedded system cited services or service level agreements, while half of the nonembedded systems did. While none of the nonembedded systems developers cited reuse of design itself, they did cite reuse of the design models, design patterns and design products. All of the nonembedded systems practitioners cited reuse of the design, five cited reuse of the design patterns, but only one embedded systems expert cited reuse of design patterns and none cited reuse of design products. This is an interesting difference. Reasons for these findings are discussed in section 5.3.

5.2.2 Coding the Answers for Quantitative Analysis

At this point, the responses were coded in an excel spreadsheet. To create the codes, an initial set of attributes had been created based on the questions, with an initial set of terms under each attribute.

The transcripts of the interviews were entered into an Excel spreadsheet sentence by sentence, ending with a total of 3462 sentences. It turned out that many sentences were further explanations of prior sentences or anecdotes to support prior sentences. These were joined to their base sentence to avoid double counting of code terms. Other sentences were about the subject's background, these were combined into a single sentence for each respondent. This left a total of 1657 codable statements.

Each statement was classified as coming from a respondent specializing in embedded systems or nonembedded systems. The one who specialized in both types of systems was coded as both embedded and nonembedded. The statements were then coded as being positive, negative or informative (P, N, I). Informative meant that the respondent was describing a concept without making a valuation of the concept. The comments were 346 positive, 285 negative, and 142 informative for embedded systems, 302 positive, 200 negative and 142 informative for nonembedded systems. 250 statements were statements about software development, many important, but not directly about reuse. Many of these statements provide insight into the reasons for some of the observations. We notice that the frequency of comments made by the respondents is not necessarily the same as the order in which they answered the direct questions. For example, while there were easily the most comments made by the respondents about components, only 63% of the respondents reported using components, whereas 100% reported reusing code, design models and requirements.

Other keywords emerged as the coding progressed. The final set of keywords is shown in the leftmost columns in Tables 5.10, 5.11, 5.12, and 5.14.

Especially in the area of success factors and obstacles, key words that did not surface in the respondents' direct answers to the questions about the attribute showed up in comments, hinting at being important. These key words were added to the appropriate attributes. One attribute that had several key words added was Technical Success Factors. We added SOA, as it was cited in 21 comments. Other key words added to Success Factors include Autogen, Similar Environment or Project, Trade off, Make fit, Parameter Driven, Comprehension, Standardization, Consistent or Controlled Baseline, Testing, Autotesting, Maintenance, Portability, Platform Independence, Platform Standardization, Searchable Library, and Documentation.

The coded responses were categorized into comments about embedded and nonembedded systems, and further into positive, negative and informational comments. The purpose was to ascertain whether the keyword was viewed positively, negatively or neutrally by embedded and nonembedded systems. From these categories, we could compare the keywords and their similarities or differences between the types of systems.

5.2.3 Results from Coding of Responses

One important element in the analysis of the coding is that the frequency with which a keyword is mentioned by the respondents, as it is an indicator of what concerns the respondent when talking about reuse. The frequencies reported below when seen in combination give us one insight into how these keywords impact the reuse experience. For example, a keyword in a positive comment can indicate that that keyword is helpful in reuse, whereas a keyword in a negative context can indicate a problem associated with that keyword. Another element of the analysis is the context in which the comments are made, which may reveal why the reuse experience is either successful or challenging.

Development Approach by System Type and Rating

Table 5.10, summarizes the number of comments about each development approach. The leftmost column contains the name of the development approach, the second and

third columns the number of positive comments made by embedded and nonembedded system respondents respectively, the third and fourth columns the number of negative comments made by the embedded and nonembedded system respondents respectively, and the final two comments show the number of informative comments made by embedded and nonembedded systems respondents.

Embedded Systems Experts in embedded systems mention a model based development approach in a positive way most frequently (16 times), product line second (10), and ad hoc third (6). These experts are positive about model based development because models are platform independent, easy to understand, easy to modify and serve as documentation. They like product lines because the systems can be implemented with minor modifications. The benefit they cite for ad hoc development is that the code is already known, understood and proven. Only one respondent mentioned an ontology approach, in a positive way. The embedded systems experts mention ad hoc negatively most frequently (11 times), with few negative comments about product line (2) and model based development (1). The problem they cite with ad hoc is that it often lacks provenance and it may have not have been developed to acceptable standards. The problem with product line development is that it can limit solution options. They did not have negative comments about component based development or ontology, and had no mention at all of SOA. Embedded systems experts gave information about model based development most often (7 times), with ad hoc second (6 times), product line (4 times) and component based development(2). They did not offer informative statements for SOA or Ontology.

It is interesting that, while more embedded systems experts report using an *ad hoc* development approach, that approach only ranks third in their positive comments. The reasons they give for success using the ad hoc approach are that they are reusing their own architecture (Person G), they are using a known function intact and exposing it to an interface (Person J) and the individual has produced the code before and knows it (Person K). The negative comments about ad hoc reuse include concerns that "the heritage is the

real issue, because much of the heritage is ad hoc" (Person D), "it wasn't finding things that was the biggest challenge, it was deciding whether they could be trusted" (Person E), and "ad hoc reuse you might get lucky and find something, but for the most part you're not going to find anything. The problem is you inherit the software irregularities or inconsistencies, if it's got defects" (Person M). According to the embedded systems developers, they are using this development approach because the project has been around a long time, and redesigning into a different development approach at this time does not make economic sense. However, for new projects, they would use a different approach, because often the older software products are obsolete, have latent defects and otherwise cannot be trusted.

Although only three of the embedded systems developers reported using *model based development*, the responses about model based development were all positive but one. Person D, an expert in FPGAs said, "model based actually helps portability because you're letting the model change your target." He went on to explain that the differences in platforms were built into the model, and the model could be used to generate the code. Person L was especially interested in using model based development. He pointed out that "Models are usually capturing information in a metamodel format, compared to raw coding statements... but certainly (models offer) the ability to define use cases, which helps you to understand the requirements. .. Then use case reuse, certainly things like diagram reuse, where a product was built to be potentially reused then the models are useful." He went on to discuss the benefits of a graphical representation over verbal descriptions and code to help a developer understand what was happening in that part of the system. His negative comment about reusing via model based development was not about technical challenges but nontechnical issues: "You can't just go and tell a program manager hey, I guarantee you will have 50% less bugs and errors if you do it this way, but it is going to cost you 30% up front. And that's what these modeling techniques do. You have to do up front work. You're spending a little up front to save a lot on the back end, but the program manager doesn't see it." There was also a large difference of opinion in what constitutes model

based development. While some contend that the use of models makes the development approach model based, others insist that to be considered model based, the development approach must include a complete set of integrated models, to include architecture models, UML and design models, performance models and simulation models. As Person K said, "I would say that model-based is all those things integrated together into some cohesive set of integrated models." Others were somewhere in the continuum.

The benefits of using a *product line* for embedded systems, according to our respondents, included, "within our product lines, basically, we started finding ways of creating framework pieces for running in the simulation environment, for running on top of hardware, for generating a common set of utilities and forcing the systems guys who were writing our requirements to reuse these pieces... there are reference architectures for these product lines, and you can do reuse along with the reference architectures" (Person G). Person M added. "When we are talking about within a product line, one of the main benefits is cost, but we also get quality and schedule, because if we are using about 90% of the build and we only need to change about 10% of the build for this new version then we are getting all of that schedule and cost reduction and we're not creating new defects in the other 90%." The product line for Person M's project is stable, and has been for many years. They deliver their products in blocks (incremental upgrade groupings), and only add a few new capabilities for each block.

Finally, one embedded systems expert indicated he was using a *component based development approach*, but said nothing further about it.

Nonembedded Systems By contrast, nonembedded systems experts mentioned positively ad hoc reuse (8 times), SOA second most often (4 times), component based development (2) with model based and product line development each mentioned positively only twice. (It should be noted that those who mentioned SOA were, in every case, using a model based approach.) The benefits cited for ad hoc reuse were that many assets exist that have already solved a requirement. The benefits of SOA are that the services are

contained and do not require developer attention. The benefits with model based are that the models are easily understood and easily modified. Nonembedded systems experts mentioned ad hoc in a negative way most (9 times), with model based and product line second (only once each). Again, the problem with ad hoc reuse is the uncertainty of the quality of the product being reused. They did not mention component based, SOA or ontology in a negative way at all. Nonembedded systems experts offered informative statements for ad hoc 13 times, model based 3 times, and product line once with no informational statements about component based, SOA or ontology.

Nonembedded systems experts again reported using *ad hoc* reuse most frequently. Person B explained the reason as, "You can inherit the software and you can actually verify that its doing what you expect it to do in the new environment. Those kinds of things drastically change the reuse factor and the ability to reuse something out of the box." Person C pointed out that "I think like almost everyone I know, my main success stories on reuse are from scavenging reuse." In other words, identifying a function or routine that would serve his purpose and developing an interface to it helped him most. This usually meant not modifying the code itself. Person H indicated that there were some standards involved in selecting the reuse product, that made the reuse beneficial: "We would understand what process did the person or organization that developed this apply to determine what was the right classification for this tool , and we would go through then and make sure it's appropriately registered and baselined." The ad hoc reuse approach worked best when the code had been properly developed and when modifications were minimal. This development approach was focused on code reuse.

Ad Hoc reuse presented some problems as well. Person B points out that "reuse has been a troubling issue in that nearly all of the models that I have seen, have used what I would describe as a "cut and run" strategy, where a piece of code or a snapshot of a particular software baseline is inherited on another system, and then that system ends up taking over, modifying it and essentially becoming responsible for a variant of that baseline and really duplicated the total life cycle cost of maintaining what now becomes

somewhat unique software." This is what the experts call "forking," which will be discussed later. However, the problem in terms of the development method used here is that of code modification. Person H mentioned that on one program he worked on using a legacy approach the software worked fine, but when they ported it to a new platform, the software broke. The code would not run properly on the new platform.

All of the experts specializing in nonembedded systems with positive comments about *model based development* approaches cited Service Oriented Architecture (SOA) as a factor in their success while none of the embedded systems experts had mentioned SOA. Person B stated the reasons most clearly: "Service Oriented Architecture is fairly liberating in that the original component itself is designed to be a reusable service rather than a point solution for a particular application, and that really has been what has been revolutionary." He went on to say, "The Service Oriented Architecture and the cloud architecture, its really the loose coupling techniques that are the enablers." Many nonembedded systems are now web based developments using a cloud architecture. The services allow the developers to treat components as black boxes. The only negative comment about the model based development approach among the nonembedded systems experts was, "So I go over to <program> and, of course, they tout model-based development, reuse there, like everything else I've looked at it appears that is an advertisement not an adopted culturally sound process." So the only negative words about model based development were that the process was not being implemented. The emphasis in this development approach is in the architecture and on the design patterns.

Product line development and *component based development* approach were not discussed by the experts in nonembedded systems.

Artifacts by System Type and Rating

Table 5.11 shows the comments made by the experts about artifacts. The leftmost column contains the name of the artifact, the second and third columns the number of positive comments made by embedded and nonembedded system experts respectively, the

Table 5.10: Development Approach ∩ System Type ∩ Rating

	Positive		Negative		Information	
Development Type	Embedded	Nonembedded	Embedded	Nonembedded	Embedded	Nonembedded
Ad Hoc	6	8	11	9	6	13
Component Based	0	2	1	0	2	0
Model Based	16	1	0	1	7	3
SOA	0	4	0	0	0	0
Product Line	10	1	2	1	4	1
Ontology	1	0	0	0	0	0

third and fourth columns the number of negative comments made by the embedded and nonembedded system experts respectively, and the final two comments show the numbers of informative comments made by embedded and nonembedded systems respondents.

Embedded Systems Reuse of components and models received the most positive comments from embedded developers (70), with architecture second (60), code third (57), design fourth (51) and requirements fifth (42). Hardware, design patterns, and interfaces were mentioned positively 24, 23 and 22 times respectively, test products 18 times, nonfunctional requirements 13 times, documentation 11 times, algorithms 8 times, test clusters and concepts each 5 times, and services once. Service level agreements and data products were not mentioned. Reasons for positive comments about components and models reflect the positive comments about component based and model based: the components were already developed, proven and optimized to the hardware; the models are easy to understand and accommodate modification easily. The reason for positive comments about architecture, design and requirements reuse is that the solution is ready to be implemented. Positive comments about documentation reuse indicate that it is easy to edit an existing, already formatted document that merely needs particulars about the system added or changed. Benefits of reusing nonfunctional requirements and interfaces are that when the artifacts being reused were developed to the same standards as the new system, the standards are already satisfied.

The most negative mentions by embedded systems developers were code (42 times) and requirements (24 times). Nonfunctional requirements and design each had negative comments 14 times, followed by architecture and hardware with 13, interfaces with 11, components and models (9each), documentation (8), design patterns (3), algorithms, and test products (2 each), COTS and test products (1 each). Services and service level agreements were not mentioned by embedded developers. Code and requirements reuse received negative comments because of problems with the code, difficulty of understanding the code, and the fact that the reused requirements (and hence, code) may not satisfy the objectives for the new system. The difficulty with nonfunctional requirements is that if the standards required by the new system differ from the old system, the artifacts need to be redone to satisfy the different standards. The problem with documentation was that it may be of low quality or missing altogether.

Informational statements about code were made most often (20 times), followed by hardware (17 times), components and models (13 each), architecture (9), requirements (8), interfaces and data products (5), nonfunctional requirements (4), design and test products (2), and concepts, COTS, algorithms, test clusters and documentation each got one informational comment. Service level agreements, services and design patterns were not the subject of any informational comments from embedded systems developers.

When directly answering the question about the artifacts used, as discussed in Section 4.1, embedded systems experts cited *code, design and requirements* as the artifacts used most often. However, as noted above, these were not the artifacts they talked about the most. Code and requirements were artifacts from the legacy systems they were migrating for reuse. Person D, developing FPGA's explained his reuse of code "Basically say I'll use the same device, I'll use the same code and then that's it." Person E added, "Requirements specifications, design specifications, was in all embedded code, the test cases, and test scripts got reused, and especially the requirements specification." So the requirements had already been implemented in code. Since the requirements were essentially unchanged, the rest of the assets could be reused. Time and money were saved in requirements devel-

opment, architecture, design, implementation, test and maintenance. According to person G, "We see a lot more reuse in the system-level design requirements, therefore we reuse the software components that were developed for those requirements. If you talk about affordability, it comes down to how many lines of code are you going to have to develop, and so the larger the item is, the more value it's going to pull forward for affordability." There were some factors that needed to be present for requirements and code to be successfully reused. According to Person K, "One (key) is to understand existing code, obviously, and software, and requirements and how its implemented in the architecture, that kind of stuff, you have to know that fairly well." Finally, Person N stated that, "I don't know that I've ever been on a program that just said let me reuse requirements or design. Typically it started as code reuse, and flexibility of that code applicability of that code being utilized in full or some modification to it for your project, then the documentation, so to speak, the design and the requirements, come with it." So the embedded systems experts start with an understanding of existing code and import the artifacts they need along with that code to develop their systems. However, as person D pointed out, as automatic code generation becomes more robust, the need to reuse code will decline.

Negative comments about requirements, design and especially code offer a great deal of insight as to why code reuse often fails. According to Person D, "most people do not write reusable code because they're in such pressure to get designs out and they also write in low level languages like assembly code. Once you start low level assembly language coding, you can't rearchitect, you're just basically patching. And so this is what we see over and over again, and is the primary reason we can't reuse or I have great reluctance to reuse any of these previous codes written. The first thing I do when I go look at it is run it through some of my analysis tools and that will tell me there's no architecture to this. People absolutely just started coding and it's a big ball of mud, so why would I waste my time with it? So it's the lack of doing a high level preparation just like the lack of documenting the requirements or making sure the requirements are comprehensive, complete, accurate, lack of testing it correctly. So that is actually probably the number

one issue to low level coding is antiarchitecture." Much of the older code was not subjected to the correct development processes, and is therefore brittle. Many practitioners would rather redevelop than try to correct these problems. Person J adds, "For me, code reuse is almost always a recipe for failure, because, for me, what code reuse says is I go in and I use some modules and I don't use other modules, in other words I'm getting into the code and I'm deciding I'm going to use this and I'm not going to use that, and modified this, and modify that I think when you start picking code apart at that level of granularity, reuse is probably, I've never seen it succeed." If the code needs significant modification, it is almost never effective to reuse it. Person L points out that, "Even with software practitioners, it takes a while to build a mental image from a bunch of lines of code. And if that's what you are presented with, you say to yourself while I know what I've got to build, and it's easier to build it myself." Without solid documentation, including graphical representations, it is easier to recreate the module than to try to understand the candidate reuse artifact.

In the embedded systems, the reuse of *models* was particularly interesting. While all of the respondents use models, it is the types of models that differ. For the embedded systems experts, the performance models and simulations were the most useful. In fact, some regenerated their code anew with each instantiation from the performance models. In many cases, their design models and their performance models were one and the same. Person D points out, "They might not use the same tools. At the high level that starts decompositioning these models into synthesizeable models. That means it can be placed into FPGA type of circuits using, give examples, it is called ESL (Electronic System Level) and it is also called HLS (High Level Synthesis). These are the techniques where you use these high level models and you use a tool to convert it directly to the FPGA code. It's based on model based engineering principles, where you stay at a high level, relative to autocoding techniques that software has been using where they develop a model and then use something that converts it to C code or that targets their processor." Person G says that, "so for embedded systems I think what you might call emulation models are absolutely critical." According to Person M, "In other cases, we actually create a

simulation of the entire module or even a set of modules to simulate how they are going to react and operate and do that for testing as well as training purposes." All of the embedded systems experts mentioned using performance models to ensure their system would meet performance requirements before moving to implementation. These models would be reused from instantiation to instantiation, with the performance models adapted to reflect any new performance requirements.

There were also a few negative comments about model reuse. Person A indicated that, "Models don't always follow the same discipline and the same elements, the same structure as the ones that you're using, so it depends on the artifact, but in each one of those, the obstacle is to make it adaptable to the particular environment." Person D said, "So some of the reused code we see is at the modeling level, at the higher level such as the Simulink blocks, that they may reuse the algorithm, the application, but we're still in the issue of how do you get that into the VHDL, into the low level code. To the FPGA." The challenge is translating the model into code for the core processor. There is also the question of whether the cost of developing a model is justified. According to Person L, "The overhead of creating a model, the overhead and the time of creating the context, the construct of the model would not be cost-effective for a small component."

Nonembedded Systems Unlike the embedded system experts, nonembedded systems experts mentioned architecture positively most (54 times), followed by code (43 times), components and models (35 times each), requirements (34 times), and design(32 times) Less frequent positive mentions went to test products and interfaces (22 times each), design patterns and algorithms (18 times each), hardware and documentation (2 times each), data products (12 times), and services and COTS (11 times each). Also positively mentioned were nonfunctional requirements (9 times), service level agreements (5 times), test clusters (3 times) and objectives (2 times).

As with embedded systems experts, nonembedded systems experts commented negatively about code (22 times), requirements (19 times), and nonfunctional requirements (14)

most often. In nonembedded systems, hardware received more negative (11) comments than design (8), and algorithms(8). Components, models and documentation received 4 negative comments each, data products had 3, test products 2 and objectives 1.

Nonembedded systems developers made informative comments about code, component and models most often (13 times), interfaces (9 times), and requirements (8 times). They also made informational comments about hardware, documentation, and data products (6 times each), algorithms, test products, and services (5 times each), design and architecture (4 times each), nonfunctional requirements and objectives (three times each) and test clusters (once).

Architecture was a primary consideration for nonembedded systems. Person B stated that "It (architecture) greatly impacts the reuse factor, because again you are standardizing the architecture and you are abstracting away a lot of the variabilities that affect reuse. So that has been a big deal to us." According to Person C, "Composeable compliance applies at all levels of our architectures, it's not a code concept it is a design concept. That kind of reuse is one in which I see a lot of promise." He went on to point out that the architecture is the foundation of this composable compliance. Nonembedded systems experts especially mentioned reference architectures as a facilitator for successful reuse. For example, Person F said, "I'm sure those reference architectures saved us a boat load of money, though we never took credit for it." Person K addressed the benefits of reusing an architecture as "When we do design we would do things to show how the components relate, what are the dependencies between the components? So we have some kind of an architecture diagram, that shows dependencies, we try to make sure that any of the designs we have do not introduce circular dependencies, we show, also, layers in the sense of what's at the top, the application level if you will, versus where's the middleware. It's just another way of showing how the applications relate, or the components relate to each other."

The only negative comments nonembedded systems experts had about reusing an architecture involved problems when architecture is inadequate.

Table 5.11: Artifact ∩ System Type ∩ Rating

	Positive		Negative		Information	
Artifact	Embedded	Nonembedded	Embedded	Nonembedded	Embedded	Nonembedded
Concepts/ Objectives	5	2	1	1	1	3
Requirements	42	34	24	19	8	8
Nonfunctional Requirements	13	9	14	14	4	3
Interfaces	22	22	11	4	5	9
Services	1	11	0	0	0	5
SLA	0	5	0	0	0	0
Architecture	60	54	13	4	9	4
Component	70	35	9	6	13	13
COTS	8	11	1	4	1	0
Data Sets, Databases, Data	0	12	5	3	5	6
Models	70	35	9	6	13	13
Design	51	32	14	8	2	4
Design Patterns	23	18	3	0	0	0
Hardware	24	16	13	11	17	6
Algorithm	8	18	2	8	1	5
Code	57	43	42	22	20	13
Test Products	18	22	2	2	2	5
Test Clusters	5	3	1	0	1	1
Documentation	11	16	8	6	1	6

* Artifacts listed in the order of phase baselined

Success Factors by System Type and Rating

The comments about success factors and obstacles were classified as technical factors and nontechnical factors and evaluated separately. In the analysis of success factors and obstacles, there were very few informative comments, so the information columns have not been included.

Table 5.12 shows the comments about the technical success factors affecting reuse. Positive comments indicate that the factor helped accomplish the reuse. Negative factors indicate either the absence or poor implementation of that factor.

Embedded Systems The most positive comments in embedded systems were about a similar environment or project (29). Standardization was second (22), with a consistent or

controlled baseline third (20). Products designed for reuse and experience were next (15), followed by test products (14), trade offs and autogeneration (13), testing (11) and a searchable library (9). Comprehension and the ability to make the products fit the system were next (8). Encapsulation, portability and platform standardization had 6 positive comments each, object orientation, platform independence and documentation had 5. Maintenance was a success factor for 3 comments. The similar environment or project means that the reused artifacts migrate smoothly to the new system with little modification. Standardization of platforms allows reused code to easily port to the new platform. Products designed for reuse are often parameter driven, so modifications are made to data and parameters and not to the underlying code. With experience, especially with the reused artifacts, the developers know where and how to make needed modifications. With encapsulation, changes made to one artifact do not affect the rest of the system.

Factors impairing reuse included a lack of design for reuse (15), lack of a searchable library (or a badly implemented library) was cited in 14 comments, failure to do quality trade off received 9 negative comments, while eight commented negatively about platform independence. Lack of a similar environment or product and lack of experience accounted for 7 negative comments each. There were five negative comments about both standardization and a consistent or controlled baseline. When products are not designed for reuse, needed modifications are difficult to make, and often lead to the insertion of new defects. The attempt to make products platform independent has a negative impact on performance.

For embedded systems, the *similar environment* was the most important success factor. According to person A, "I would say the success is driven by how similar the new environment is. If you were doing manufacturing or cookie cutter things, where you could just pick up a something and reuse it, then you would be very successful." The less the product needed to be adapted, the easier it is to reuse. Person D added, "I think that reuse and commonality go together." According to Person E, the similar environment is especially important in terms of platforms and interfaces: "Things like, especially for an embedded system, processor hardware and interfaces. To see if they are at least simple,

if not identical." According to person G, "a piece of software will use somewhat the same utilities, will use somewhat the same way of communicating, and somewhat the same way of attaching into the system...And then if you can choose a receiver that is similar to a receiver that we've already used, an exact one or one that uses a similar protocol, you can get even more reuse out of it." Person J points out that "the use cases don't change, the fundamental architectures don,ÄôÂôt change, requirements don't change at a very high level, they do at a very fine-grained level, the basic requirements are there, the basic test patterns are there, the basic architecture is there, so what you end up doing is fine tuning down at the code level."

When there was not a similar environment, Person A said, "I think that if it (the environment) is very dissimilar you'd better be careful." In the words of Person D, "The reason we don't have commonality is because everybody redesigns everything over again. Same issues with reuse. It's (similar environment) extremely important." Person G suggests that lack of a common environment almost prevents reuse. "If you come along with a system that has something different, then of course they don't have much in terms of reuse." He also points out that one of the problems with a company or corporate reuse mandate is that environments are not the same. "In fact it's that part of the problem that anyone wants to start up a reuse workgroup or try to make a business unit reuse strategy, what they want to do is have a one size fits all. And that's impossible. And so it basically starts off as something where they are going to collect a certain set of artifacts and force everybody to use them, and you get all the different types of products that we developed together they all absolutely refuse. So it falls down pretty quickly. You have to recognize the with the business unit this size is that it's not a one-size-fits-all. So any centralized reuse, then, would have to take that into account." Person N adds, "where you are trying to apply some other ones baseline into your program, that reuse didn't work out too well, particularly in my case, where reuse was very low, around 5%." These comments indicate that in embedded systems, reuse does not migrate well across technologies.

Standardization was of great concern to embedded systems experts. While platform standardization was universally a concern, they also mentioned standardized interfaces, standard reference architectures, verification techniques, documentation and modeling techniques. According to Person L, " My reuse, quickly formed on where you have something that is built to a standard, or a commercial product that has live utility, that's where I've seen reuse... when I have seen it (successful reuse) on projects it's been through the stuff that has a well-defined commercial or GOTS product built to a standard, because then, the thing about a standard is you know what the inputs are and what the outputs are, you know what the functionality is, and then you're using it as a whole and then interfacing to it and the goes intos and the goes outofs and following the rules, that's where it has succeeded." About standardized modeling techniques, he said, "Wouldn't it be so much faster if I had this in a modeling format that I could assimilate and then change the model to accommodate what it is I'm trying to do? Or take, then, the assurance case, and I have an assurance case for my real time operating system in combination with a messaging protocol, in combination with whatever else and build the aggregate of the models together to layer up and build my assurance case and aggregate that for the entire system." Person N added, "if we develop this under these standards now, we can save half to two thirds of the money in 5, 6, 7, 8 years. If that's done in a standard way, interrupts are all happening the same way across single board computers, the operating systems do it that way and we start doing real-time middleware, and we provide APIs to applications, and this and that, there might be details in the kind of data that is passed, but in general the methodology to go do that then you could see some standardization across stovepiped programs and maybe across industry in the way that happens." By building products to standards, developers could adapt the APIs and leave the code untouched, or modify the models with the details of platform and use the models to generate the code.

Nonfunctional requirements were especially cited for lack of standardization, requiring significant redesigning and rewriting of code. Nonfunctional requirements are further discussed under "obstacles" below.

A *common and/or controlled baseline* was also cited as an important factor in success. According to Person L, "you want to keep everything as consistent and common as possible and do an architecture and evolve the architecture based on the fact that these changes will occur." Controlling the baseline allows modifications to occur in an ordery fashion, and developers will know what those changes are and how their own development is impacted.

Design for reuse was considered an important success factor in embedded systems. According to Person D, "Number one thing is reuse is only as good as the original design and that means to do the reuse the design had to be developed originally for reuse." Person E recalls, "On one program we made the upfront investment, to design for reuse, it was an embedded system, and we sold three follow-on jobs based on reusing much of the software. The software cost in the three jobs were in the same ballpark, so net we saved ourselves and the customers money." The negative comments about design for reuse revolved around reason software is not designed for reuse. Person D says, "most people do not write reusable code because they're in such pressure to get designs out." According to Person G, "It does cost more to develop for reuse than not, with all artifacts and things. And things go beyond what your program's actual need is." Several embedded systems experts indicated that design for reuse costs around 30% more than regular single use design. To justify this extra expense, there needs to be a business case that the resulting artifacts will be reused enough to justify this added expense. In the rapidly changing technical environment, there is often doubt about sufficient reuse of a given artifact. In addition, there is a question about paying for that extra investment. In bespoke projects, the customer does not want to pay extra in order for the next customer to benefit. We note that design for reuse is almost always discussed in terms of code.

Nonembedded Systems By far, the most common positively mentioned success factor for nonembedded systems was a consistent or controlled baseline (29). Second was SOA (18), and test products/autotesting was third (17). Standardization had 15 positive comments, testing and platform standardization each had 14. A searchable library had

12, encapsulation had 11 and platform independence had 10. Comprehension and design for reuse each had 8. Documentation (7), similar environment or project (6), project (6), ability to make it fit (6), and experience (5) came next. Autogeneration of code had 2 mentions, as did maintenance, and object orientation only had 1. As with embedded systems experts, nonembedded systems experts cite the consistent baseline because developers knew the baseline they were working with, and did not have to keep up with constant changes. SOA was considered a success factor because the modules were self contained, and only the interfaces needed to be accommodated. Nonembedded systems experts cited a searchable library because often they found it difficult to locate and understand products that they could reuse.

The most negative comments about success factors in nomembedded systems were about platform independence, and a searchable library, with 6 apiece. Design for reuse and a consistent or controlled baseline each had 4 negative comments. SOA, standardization, maintenance, portability and experience each had 2. Autogeneration of code, similar environment or project, trade offs, making the product fit, object oriented development, test products, and platform standardization each had 1. The rest of the factors were not mentioned. As with embedded systems, attempting to attain platform independence was seen in a negative light because the disparity of platforms made abstracting the interface to accommodate multiple platforms difficult, and often the interface impacted performance.

Nonembedded systems experts were most concerned about a *consistent or controlled baseline*. Person B points out that " We have one example within our domain of essentially a cloud, elastic cloud, that we have maintained the discipline of having one baseline across multiple programs and maintaining that one baseline. And so when another variant, when another program uses it they have the responsibility (sort of like the open source model) they have the responsibility of migrating those changes back into the single controlled baseline Now, of course, that's easier these days as the environments themselves get more and more standardized. You can wrap all of the configurations, all of the setup into a virtual machine image that really is just a bundle of all the operating system, the COTS

software, and then the applications running on top of that stack." A consistent, controlled baselines that is properly encapsulated allows components to be changed or added without impacting other components. Another success factor for nonembedded systems was the use of a SOA, which was discussed earlier.

Table 5.12: Technical Success Factors ∩ System Type ∩ Rating

	Positive		Negative	
Development Type	Embedded	Nonembedded	Embedded	Nonembedded
Autogen	13	2	0	1
SOA	1	18	0	2
Similar Environment or Project	29	6	7	1
Trade off	13	2	9	1
Make fit	8	6	2	1
Object Oriented	5	1	0	0
Parameter Driven	4	3	0	0
Comprehension	8	8	2	0
Design for Reuse	15	8	15	4
Encapsulation	6	11	0	0
Graphical Design	1	0	0	0
Test Products/ Autotesting	14	17	2	1
Standardization	22	15	5	2
Consistent or Controlled Baseline	20	29	5	4
Testing	11	14	2	1
Autotesting	2	3	0	0
Maintenance	3	2	0	2
Portability	6	8	2	2
Platform Independence	5	10	8	6
Platform Standardization	6	14	0	1
Searchable Library	9	12	14	6
Documentation	5	7	1	3
Experience	15	5	7	2

* Negative comments about success factors indicates that that factor was missing and led to problems with the reuse.

Nontechnical success factors, shown in 5.13 were analyzed separately from technical success factors, because they require a different type of solution.

Process and control/discipline were nontechnical success factors mentioned most by embedded systems (25 and 20). Other factors were planning at 10, management commitment at 7, culture at 2 and estimation at 1. The most negative comments concerned process (16), management commitment (14), and cost/schedule estimation (11). Negative comments were also made about control/discipline (6), culture (3) and planning (4).

For nonembedded systems, the most commonly mentioned nontechnical success factors were process (21), planning (16) and control/discipline (15). Culture accounted for 10, management commitment for 6 and estimation for 3. By far, the most negative comments in nonembedded systems were made about etimation (15). Process (9), control/discipline (8), and management commitment (7) were close. Culture was mentioned twice. Planning was not mentioned negatively.

Table 5.13: Nontechnical Success Factors ∩ System Type ∩ Rating

	Positive		Negative	
Success Factor	Embedded	Nonembedded	Embedded	Nonembedded
Management commitment	7	6	14	7
Process	25	21	16	9
Control/ Discipline	20	15	6	8
Culture	2	10	3	2
Planning	10	16	4	0
Estimation	1	3	11	15

One interesting negative success factor comment was when management mandates reuse but does not provide the necessary tools to identify and analyze the candidate products. Person C said, "So for instance, there was a time when we had to use the reuse repository, or explain why we didn't use anything. That, to me, it's like government taxation. You are encouraging certain behavior and discouraging other behavior but ultimately, what it did to me is make everything more expensive. That was a non-benefit of reuse. When it came in mandated to me, it was going to cost money whether I reused it or not, and if the repository had been at that time in the shape I really could have saved money every time I used it it would've been a good application of the tax, right? It would've encouraged me to perform the behavior that was beneficial to the company. Since the intent was to encourage me to do it that was fine, but in fact the repository was not in good enough shape at that time to benefit more often than simply have to prepare a report that would take some number of hours of a senior engineer's time. So it ended up being a tax."

Obstacles by System Type and Rating

As with success factors, obstacles were classified into two categories, technical and nontechnical. Table 5.14 shows the technical obstacles to reuse success, Table 5.15 shows the nontechnical obstacles.

We discuss the obstacles to reuse in the opposite direction from the success factors, negative before positive. In situations where the obstacles have positive comments, the obstacles were overcome, usually by a success factor.

Table 5.14: Technical Obstacles ∩ System Type ∩ Rating

	Positive		Negative	
Obstacle	Embedded	Nonembedded	Embedded	Nonembedded
Fit	5	1	22	13
Modification	10	6	26	33
Understanding	13	3	27	15
Complexity	6	2	3	9
Lack of trust	1	0	4	0
Lack of Documentation	0	1	10	3
Lack of Metrics	2	2	9	4
Obsolescence, age	2	1	8	7
Easier to build from scratch	0	0	13	5
Platform Dependence	3	0	17	10
Loss of flexibility or freedom	0	0	6	8
Difficulty - real or perceived	0	0	14	11
Unintended Consequences	0	0	4	10
Missed Opportunities	0	0	11	7
Existing Defects	1	0	13	3
Insert defects	0	0	0	3
Performance Goals	2	5	5	6
Maintenance	0	4	2	11
Certification Process	0	4	3	3
Forking	1	0	5	8

* Positive comments about obstacles are coments in which the obstacles were overcome.

Embedded Systems Understanding was the greatest obstacle to successful reuse in embedded systems (27), followed closely by the need for modification (26). Embedded systems experts cite obscure code and poor or no documentation as reasons for the obstacle of understanding. Twenty-two comments indicated that reuse candidates turned out to not fit the system they were being integrated into. Here, the experts explain that the artifacts

appeared to be a fit until they were to be integrated into the system, and the interfaces or approach were incompatible with the new system. Platform dependence was considered an obstacle in 17 coments, again reflecting on the difficulty of moving code optimized for one platform to a platform for which it had to be reoptimized. Real or perceived difficulties were seen as obstacles in 14 comments. Thirteen negative comments indicated that it was easier to build from scratch, the same number found that existing defects were a detriment to reuse. Experts indicated that in the time and effort required to become familiar with the reuse product and/or identify and correct existing defects they could create a new product. Failure to reuse because of missed opportunities were cited in 11 comments. Embedded systems experts indicated that they did not reuse some products they could have reused because they either did not know they existed or did not trust the artifacts. Lack of documentation (10), lack of metrics (9) and obsolescence (8) were also problematic in embedded systems. Loss of flexibilitly was a problem mentioned in 6 embedded systems comments, meeting performance goals and "forking"[1]. Lack of trust of the reuse candidate and/or its developers and unintended consequences were cited four times as obstacles, complexity and the certification process were cited by three.

Several embedded systems respondents mentioned overcoming obstacles. Overcoming the obstacle understanding was mentioned in 13 comments and modification was mentioned in 10 comments. Six comments mention overcoming complexity and 5 the problem of bad fit.

For embedded systems experts, the biggest obstacle was *understanding the reuse artifacts*. This happened most frequently with code. Person D describes his response when given a code reuse product as, "How do I interface it? Am I sure that the verification that came with it adequate? Is the documentation adequate, do we understand it enough that we can modify it?" If the answers are not in the documentation, even if in the end the code would not require modification, he states that it often takes longer to figure out these issues

[1] "Forking" is defined as taking a reuse product from one project and modifying it for another without coordinating the modifications with the original project. This is different from lack of configuration management in that the new project will keep the forked artifacts under their own configuration control. The problems with forking are discussed in the analysis.

than to simply develop from scratch. According to Person K, "A lot of times it's a misfit when people who quoted the reuse maybe did not really dig deep enough to understand how much the design and the requirements, and of course, then the code, would have to change. Then one of the biggest problems is incorrectly reusing things that aren't quite right, and being too optimistic in thinking you didn't have to change anything, when you actually did have to change it. That is a major hurdle." So accepting this reuse product and trying to work with it when it was not quite right ends up costing more than new development. Often projects struggle with a reuse product until they have invested a great deal of effort in it, only to have to discard that work and start over. Person K adds that, in addition to knowing the product being reused, "you have to fairly well know the system that you are actually trying to put it into, what that needs to do, and what those requirements are, and key aspects of that system.... I would say the barriers are pretty much, there's a few other reasons, but those are the key things that, if you cram something in there that isn't a right fit because you didn't understand either what the reuse did, or what the system you want to put it in had to do, then you're going to fail." Finally, Person N points out that often a reuse candidate is rejected because, "I don't know it, I don't understand it, I would have to make too many changes, it would take me too long to understand what it did, all of those reasons." So for embedded systems, either reuse is not attempted or it fails because of lack of understanding of the reuse artifact.

The *need for modification* was another obstacle to reuse. Person A describes failure in reuse as, "software that doesn't work picking it up and using it in your environment and you cannot modify it cheaper or faster than you could have developed it from scratch. If the reuse is not going to meet your performance requirements, then you have to, again evaluate well what do I have to do to that reused software so that it will meet my performance requirements, and if that means I have to go in and redesign it and rebuild it then it may not be worth the reuse." Person D's experience with inheriting a product was, "As soon as they tried to do a slight modification to this (code), the whole thing fell down. It's like all of a sudden we found out it was all patched together, we found out that you might as well

throw it all away and start over." Having undergone this experience, he became reluctant to attempt to reuse code if it required any modification: "if you want to do changes, you want to modify it, the code is not written in a readable, modifiable manner and you get in all kinds of trouble." Person J adds, "So for me, code reuse is almost always a recipe for failure, because, what code reuse says is I go in and I use some modules and I don't use other modules, in other words I'm getting into the code and I'm deciding I'm going to use this and I'm not going to use that, and modified this, and modify that I think when you start picking code apart at that level of granularity, I've never seen it succeed." The reasons for fit being a major obstacle are similar.

Platform dependence was a major obstacle for embedded systems. Every embedded systems expert stated that code reuse becomes a problem when the platform changes. FPGAs have a particular challenge with platform portability according to Person D: "That is you can write code to be portable between platforms but because each of the platforms we're using is a system on a chip now. That means they have DSP functions in them, high performance mathematic devices, they have so many structures of memory, dual port, quad port you name it. Inside you have gigabit serial links. If you are trying to write portability from one device to another device, it may not work because each device has its own set, its own difference. So you can write to be generic and portable for some but when you get to the main elements of a chip, for example, the new Xilinxs chips, zinc, have an arm 7 or is it an arm 9, a very high speed processor inside it. So there's obviously no portability there, so the portability is kind of a dog chasing its tail." Person J added, about FPGA platform dependence, "I'll give you an example, one that was not in the too far distant past, we had a whole bunch of FPGA code for reuse So a lot of the VHDL that's written for one family of FPGA versus another and we end up rewriting a lot of the VHDL as we move from one family of chips to another. So VHDL does not equal VHDL. It has to do with the libraries for those particular chipsets. So that's one of the considerations." Again, this issue is with code. Person G also had issues with changing platforms. He pointed out that, "if the next embedded system had a different processor,

we may not have ended up using a lot in the way of designs, we may or may not even have the same computer language so even if you did have the same computer language, differences with compilers, development environments, sometimes meant that you had to mess with the code. Especially for systems that are doing primarily control and status of hardware, for some of those systems if you change the underlying hardware including the CPU, processor, and hardware that interfaces to the digital hardware, you might have to redo 80% of your code even if the functionality is totally identical, if you change the hardware." Person G has another reason for platform dependence being an obstacle to reuse, and that is in the speed at which hardware is changing. He says, "because right now what we have in the embedded world especially is were having a lot of flux in the processor world." He goes on to say that the hardware is changing faster than the software. Obviously, reused software is going to be older than the new hardware.

Several embedded systems experts cited their *lack of awareness or inability to find artifacts* as an obstacle to reuse. As Person A pointed out, if you don't know they are there or can't find them, you can't use them. Person K stated, "one problem is the access to the information to know what there is to reuse and where it is. What actual requirements it matches, and what the design is, and the problem with information sharing is huge." Person L added, "how do you quantify, how do you find, how do you locate, then how do you really determine that this thing out there has the capabilities and the functionality and the attributes that you need to truly solve your problem? It's really not that the stuff's not out there, it's just the ability to somehow properly catalog and put together some standardized meta-model." All suggested that a searchable library with full documentation would be useful.

Finally, the embedded systems experts were concerned about the *lack of metrics* for reuse. In the words of Person D, "Hardware has none. FPGA has none. They don't want to know how bad they're doing. They know. The only way you can find out how bad they're doing is to they all say, yeah, it was all done in 6 months, you go back to the software person they say, yeah, we debugged that for 2 years in circuit. We had to write software

to work around that glitch that we could never find. So it wasn't designed in 6 months. So there are no metrics. I think they have to know deep down it's pretty poor. You know, assembly language. So the metrics are something we're pushing right now. Metrics is the basis for all process improvement." According to Person G, "I don't know that anybody has collected any metrics here, I don't know of any available out there. I think the defects go down but I don't think we've ever measured any that would quantitatively show that reused code is less defective than regular code."

Nonembedded Systems By far, the largest number of negative comments about obstacles to nonembedded systems was modification (33). Understanding was next (15). The challenge of fitting the artifact into the system was mentioned 13 times. Difficulty, real or perceived, and maintenance were mentioned 11 times, platform dependence and unintended consequences 10 times, complexity 9, forking and loss of flexibility were each mentioned 8 times, obsolescence and miss opportunities 7 times, and meeting performance goals 6 times. Five times, the comment was that it would be easier to build the product from scratch.

Modification was a major obstacle for nonembedded systems reuse. The nonembedded systems encountered two different challenges for modifying software. One challenge was the actual modification and the other challenge was the fact that the modification they could do was limited by restraints on modification given the desire to maintain a common baseline. In the former case, Person C states, "I guess the third piece of this I've run into from time to time is how easy is it for me to try to reuse an artifact and potentially then to bail on it if my initial trial, say 90 days or whatever, does not prove satisfactory. Then I'm stuck. I will need to invest in maintaining that artifact or adapting it. As you know, the costs of any project are driven by the decisions you make in the first few months. Therefore you have to make good enough decisions, so that you can live with the consequences." Person F adds, "So you get to a point where almost certainly when you are modifying about 2/3 of it, you're losing money, you would have been better off to start

from scratch." Person H states, "there were some unique requirements for our program that caused us to have to maybe do additional changes that we maybe didn't anticipate upfront." The artifact here is usually code. In the latter case, according to Person B, "the more people that you have reusing a component the less freedom of movement you have for any individual stakeholder in that reuse." In this case, any artifact could be involved.

Understanding was also an obstacle for nonembedded systems reuse. Most of the concerns were the same as for embedded systems. As Person B points out, "part of the up front cost is learning what the other guy did." However, nonembedded systems were also concerned with making their products useful for more than their own projects. Person B states that, "I would like to say that reuse in our domain is black and white, but there's always shades of gray in there where you thought you were designing something that was general purpose, but you hadn't considered a particular use case that would then need to be accounted for when it's used on another program."

As with embedded systems, nonembedded systems reuse was impacted by *platform dependence*. Person G offers an example: "One experience I can cite is I adopted an entire application. That application was produced for and ran very well on Intel CPUs and ran on AMD CPUs, however the application underlying the physics of the application was poorly adapted to an AMD. It would have required significant investment." According to Person H, "We've had issues with platform independence and how independent everything can really be. About anytime we're changing hardware we are also changing operating systems. Most recently, obviously, we had the Linux port to deal with. Not everything was perfectly, at least as it relates to the operating system. We had to make a fair number of changes to address differences between Solaris and Linux." Person I found that, "One of the problems we ran into in the past with scientific algorithms is if we move from one type of the processor to another, all the underlying mathematical libraries changed. And so the porting cost was high. But if we all agree on using Intel X 86 and the associated libraries that come with it then porting is much simpler." While most of the nonembedded systems were attempting to develop platform independent software, in some cases this was

difficult. As with embedded systems, all agreed that platform standardization would solve many of these issues.

Metrics was also an issue for nonembedded systems. As Person I put it, "We still depend too heavily on SLOC counts. In the days when everything was written in Fortran or C, SLOC counts maybe made some sense, but now that you are doing stuff in C++ and you are using commercial products, and you are using Java, I'm not so sure those SLOC counts make a lot of sense as a basis for cost estimation." Person H said, "We need to look at how we could maybe more effectively measure whether reuse really helped us or not. Because I don't have any numbers to back it up."

Table 5.15: Nontechnical Obstacles ∩ System Type ∩ Rating

	Positive		Negative	
Obstacle	Embedded	Nonembedded	Embedded	Nonembedded
Commitment	3	0	5	3
Contracts	0	0	11	3
Individualism	0	0	10	6
Culture	1	0	18	6

Fewer nontechnical obstacles to reuse success were identified, but they were often more intractable. Embedded systems negatively mentioned culture most often (18), followed by the way the contracts were written (11), developer individualism 10 times and management commitment 5 times.

Both individualism and culture were seen as problems for nonembedded systems, mentioned negatively 6 times each. Management and contracts each received three negative comments.

5.2.4 Interpretation

Here we look at the answers to see if there are differences between embedded and nonembedded systems. As mentioned earlier, the number of times the experts mention particular key words appears to be related to the importance of those key words. We

conclude this because when a key word is mentioned over the course of several questions, that key word is related to many aspects of reuse. In other words, if, when asked about practices, success factors and obstacles, the respondent uses "code" in the answer, then we conclude that code is an important element of practices, success factors and obstacles.

Similarities and differences between embedded and nonembedded systems overall in terms of reuse Both embedded and nonembedded systems had more positive comments than negative comments about reuse, however, the ratio of positive vs negative was much less for embedded systems by a 6:5 ratio, whereas nonembedded systems was by a 3:2 ratio. All respondents reported reusing code and requirements, regardless of system type, except for one nonembedded system developer. All but 2 of both the embedded and nonembedded systems used test products, which include such things as test drivers, test scripts, and test cases.

All embedded systems experts reported reusing design, whereas none of the nonembedded systems experts reported reusing design. Rather, the nonembedded systems experts were more granular in reporting reuse in the design phase. All but one of the nonembedded systems experts reported using design models. None of the embedded systems experts reported using design products, whereas all of the nonembedded systems experts did report using them. Design products for nonembedded systems include such things as architecture framework diagrams and flow diagrams, with UML or SySML as their design models. Five embedded systems experts reported using Simulink models as their design models. All but one of the nonembedded systems experts reported reusing the architecture, but only two of the embedded systems experts did. Whether this difference was cultural, in that nonembedded systems developers think of design in terms of a set of artifacts, whereas the embedded systems developers consider design to be a whole is not clear. It may be that in their reuse process, the embedded systems developers import the code as predesigned, predeveloped units. None of the nonembedded systems reported using components, whereas

five of the embedded systems used components. A deeper look at the artifacts reused shows that embedded systems reported reusing components such as interrupt controllers, clock domain processing, and IPCore. These components would be used intact. None of the nonembedded systems experts reported reusing such components.

Differences and similarities in development approach A challenge in comparing development approach is in the wide variation of the definitions of the approaches. As mentioned earlier, defining what constituted model based development differed from a complete set of integrated models across the life cycle to the use of models in some phase or aspect. As a result of this ambiguity, the use of model based development may be overstated or understated.

A similar challenge exists in the definition of a *product line*. Person F stated that to be a product line, the company needs to build large quantities of the product, which doesn't happen with very large systems: "but how often do we in <company> build 200 almost identical copies with slight and continuous variations of a product?" Yet Person M says that to productize a system, a project will "start at requirements and make whatever modifications needed to that to add functionality, and then of course modify a little bit of the design and the code and whatever else, and add new test procedures. But we also do regression testing and make sure that all of those things still work." Person G says, "So if you are doing a product line, you really want someone in control, you really do want some central piece that is designing the system and forcing the splinters to line up along with the product line." In other words, it is the baseline control that differentiates ad hoc reuse from product line development. Again, these differences in definition could cause overcounting or undercounting of the use of product line development approach depending on the definition used.

While over half of the embedded systems developers were using an *ad hoc* reuse approach, the negative comments about this approach exceeded the positive comments by almost double. Two thirds of the nonembedded systems were using an ad hoc reuse ap-

proach, yet there was only one more negative comment about this approach than positive. This could indicate that the development approach was selected for reasons other than reuse and was the better approach. However, it is difficult to think of a reason other than reuse to use this type of approach. Another explanation could be that while there are difficulties with the reuse in this approach, the difficulties are outweighed by the alternative of starting anew with a different approach. The fact that many of these projects have been ongoing for many years would lend credibility to this explanation.

The common use of *component based development* in embedded systems as opposed to the lack of component based development in nonembedded systems could be the result of the artifacts reused. As mentioned earlier, many items identified as reused artifacts are components reused intact. Code for these components often comes with the purchased platforms and is optimized to these platforms. It is difficult to tell whether the requirements were dictating the code and platform or the other way around, which, from the comments, seemed more likely. Nonembedded systems did not report reusing components with purchased pre-optimized platforms.

While only three embedded systems experts described using *model based or product line* development approaches, the positive comments for model based were 16, compared to only one negative comment, and positive comments for product line were 10 compared to only two negative comments. This would indicate that when they can either begin their projects with model based development or convert their inherited systems to models they have a more positive experience. Person B said, "If we were starting from scratch, especially with modern techniques, we would probably do a lot more model based architecture and design, but right now, most of the significant changes are understood changes to an understood system." Thus, this expert expected that if the project was to be started today, the development approach would be different. However, the cost of changing over now would be higher than the benefits realized. Similarly, when they can productize their inherited systems, the experience is more positive. In nonembedded systems, other than the product line, none used the same development approaches. However, it could be argued that the

one reporting a SOA approach was also using both component based development and model based development.

Table 5.16 shows the ranking the comments of development approaches comparing embedded to nonembedded systems. We see that model based development receives the most positive comments for embedded systems, whereas ad hoc reuse receives the most positive comments for nonembedded systems. This is particularly interesting, given that when asked what development approach they were using, four of the embedded systems experts were using ad hoc development, and three were using model based development.

We also notice that ad hoc reuse had the highest incidence of negative comments for both embedded and nonembedded systems. This, coupled with their statements reported earlier, suggests that some of the embedded systems professionals were not using the development approach of choice. Person G had some important things to say about this: "So when we create software here there is no strategy for creating reusable pieces only for when the program starts off and for going off and examining whether reuse is available." and Person L said, "So when we create software here there is no strategy for creating reusable pieces only for when the program starts off and for going off an examining whether reuse is available. Yes, ad hoc reuse in that you might get lucky and find something, but for the most part you're not going to find anything. But I'd say the kind of reuse we get here is more or less ad hoc because there isn't anything driving it. It is done by the people putting together the system going off and seeing what they can grab, and it is ad hoc. There's no catalog, there's no help, there's no guidance, other than thou shalt do some reuse." Person L pointed out that "It's more of an ad hoc, we've got something we can reuse, those tend to be less successfully reused." Person L also said, "Plus we would be bridging technologies across a decade or more and the technology in hardware and software is evolving so rapidly that who would want to reuse something that's 10 years old? Why would you want to try to reuse a desktop that's an Intel Pentium 66 MHz machine or would you rather go by the latest one out there? That's part of the problem. In avionics and electronics, who wants to reuse something that was designed 18 to 20 years ago¿' Yet many of these projects have

to do this because they have been ongoing for several years. As mentioned earlier, if they were to start their projects today, several suggested that they would use a model based approach.

So why were they using ad hoc reuse if they would prefer to use something else? The projects using ad hoc reuse have been around for decades. The systems are not being developed as much as they are being enhanced for subsequent iterations. The cost of refactoring the systems into a different development approach, most likely model based, could not be justified. As mentioned earlier, if they were to start their projects today, several suggested that they would use a model based development approach.

Table 5.16: Ranking of Development Approach by System Type

	Positive		Negative		Informative	
Rank	Embedded	Nonembedded	Embedded	Nonembedded	Embedded	Nonembedded
1	Model Based	Ad Hoc	Ad Hoc	Ad Hoc	Model Based	Ad Hoc
2	Product Line	SOA	Product Line	Product Line	Ad Hoc	Model Based
3	Ad Hoc	Component Based	Component Based	Model Based	Product Line	Product Line
4	Ontology	Product Line			Component Based	
5		Model Based				

As we look at the actual comments from the respondents, we find that they were not very interested in what their development approach was called. As they were able, they incorporated whatever portions of the approaches that would work for them and didn't bother to give it a name. As one respondent said, "it is whatever the customer wants to call it."

Differences and similarities in reused artifacts Table 5.17 shows that there are similarities between embedded and nonembedded systems. Four of the five artifacts that get reused are the same. The only difference is small, design is fifth with embedded systems and sixth with nonembedded systems. Differences occur for less reused artifacts. Hardware is the seventh most commonly used artifact in embedded systems, whereas it is 11th for

nonembedded systems. Nonfunctional requirements are eleventh for embedded systems, but 16th for nonembedded systems. Service level agreements were not mentioned at all in embedded systems, they were 17th in nonembedded systems. Data products were 13th for nonembedded systems and not mentioned at all in embedded systems. Clearly, the artifacts reused between the two types of systems after code, components, requirements, design,models and architecture are different.

Code, requirements and nonfunctional requirements were first, second and third respectively in negative comments for both embedded and nonembedded systems. Experts from both types of systems indicated changes had a serious impact on modifying the code. However, both embedded systems and nonembedded systems experts stated that when the requirements were contained in models and the code generated from those models, the changes were fairly easy to implement. We conclude that code and both functional and nonfunctional requirements are problematic when trying to migrate code from one system to another in either type of system. Yet code, requirements and nonfunctional requirements were all three in the top five of the reused artifacts. This may be one of the reasons reuse has not lived up to expectations.

While embedded systems experts rarely commented on *architecture*, architecture was the most prominent topic among nonembedded systems experts. Person E suggested that early in the life cycle, certain components and code products were selected, and the architecture built around those items. We conclude from this that reuse of these artifacts are to some degree dictating the solution.

The statements about *models* made it clear that the respondents were talking about different types of models. We separated the types of models into four categories, architecture models, design models, performance models and simulations. Table 5.8 shows the different models and the number of comments about those models made by the respondents.

The embedded systems experts made far more informative comments about models than the nonembedded systems experts (92 vs 54). Embedded systems experts referred most often to design models, and far more often than nonembedded systems (32 vs 21),

followed by architecture models (24 and 16) and performance models (18 and 23). One big difference is the number of references to simulations for embedded systems vs nonembedded systems (24 vs 9).

Table 5.17: Ranking of Artifacts by System Type

	Positive		Negative		Information	
Rank	Embedded	Nonembedded	Embedded	Nonembedded	Embedded	Nonembedded
1	Component	Architecture	Code	Code	Code	Components
2	Models	Code	Requirements	Requirements	Hardware	Models
3	Architecture	Components	Nonfunctional Requirements	Nonfunctional Requirements	Component	Code
4	Code	Models	Design	Hardware	Models	Interfaces
5	Design	Requirements	Hardware	Design	Architecture	Requirements
6	Requirements	Design	Architecture	Algorithm	Requirements	Data Products
7	Hardware	Interfaces	Interfaces	Component	Interfaces	Hardware
8	Design Patterns	Test Products	Component	Models	Data Products	Documentation
9	Interfaces	Design Patterns	Models	Documentation	Nonfunctional Requirements	Services
10	Test Products	Algorithm	Documentation	Interfaces	Design	Algorithm
11	Nonfunctional Requirements	Hardware	Data Products	Architecture	Test Products	Test Products
12	Documentation	Documentation	Design Patterns	COTS	Objectives	Architecture
13	Algorithm	Data Products	Algorithm	Data Products	Algorithms	Design
14	COTS	Services	Test Products	Test Products	COTS	Objectives
15	Test Clusters	COTS	Objectives	Objectives	Test Clusters	Nonfunctional Requirements
16	Objectives	Nonfunctional Requirements	Test Clusters		Documentation	Test Clusters
17	Services	Service Level Agreements	COTS			
18		Test Clusters				
19		Objectives				

Differences and similarities in technical success factors Table 5.18 compares technical success factors between the types of systems. Positive comments mean that the factor was present and a reason for successful reuse. Negative comments mean that the factor was absent and as a result the reuse experience suffered. Here we find greater differences. For embedded systems, the success factor with the most positive comments was a similar environment or project. It was 14th for nonembedded systems. This implies that similarity is not as critical for the nonembedded systems.

Experience ranked fifth for embedded systems, but only 16th for nonembedded systems. From Person F, "I know in our image acquisition system the analysis, looking at the four ways to do it, the trade-offs, we had meeting after meeting with all the experts, and we all looked at each other at the end of the meetings and you know, there's only one real choice here, given our situation and constraints." Having the experienced people there helped make the right decisions. According to Person G, "The code that we reuse is internal, we have access to it, we know the people who wrote it, they can tell us the worms about it upfront and then we can say this is good or is not good, and so we don't really have many times were we picked up a piece of code and it turned out to be a disaster." People familiar with the systems protects against unsuccessful reuse situations.

Autogeneration of code was 7th for embedded systems, but only 19th in nonembedded systems. This could be explained by, as mentioned above, embedded systems frequently using a modeling tool that autogenerates code.

On the negative side, a lack of *design for reuse* and *searchable libraries* were both cited in the top five for both embedded and nonembedded systems. In the case of design for reuse, several experts said they experienced overruns because the products they were given to reuse were not designed for reuse. This is particularly a problem, because, according to two experts, designing for reuse costs 30% more than normal design and development. In bespoke projects, the customer does not want to pay that extra amount so that some other customer can benefit. On the other hand, the company needs a good business case that the products will be reused enough for the company itself to underwrite the added expense.

This usually does not happen. There are two types of problems related to searchable libraries. The first is that there is no searchable library to use. The second is that the library is poorly organized, documentation is not included with the artifacts, leading to an expectation of reusability in a given project that is subsequently proven false.

Lack of experience can create problems. According to Person D, "The problem we've seen, is when I'm called in is after some program already made their decision and the poor designer that had to inherit this, the guy that did it maybe 5 or 6 years ago was long gone or won't help or doesn't remember anything. The people that were the go to people before aren't producing the same parts anymore."

Differences and similarities in nontechnical success factors Table 5.19 shows the ranking of the nontechnical success factors by system type. The positive comments for the top three are the same, in a different order. Both show process as the most important, with control/discipline and planning also mentioned.

It is when we get into the actual remarks that we find out how important the respondents think *control* is. Person J says, "I am more authoritarian about tracking reuse. Much more. I will never allow, well we're going to do this percent of reuse and say hey guys we're going to reuse these things and then let the teams go off and do design. Never. Won't do it. Once I decide on a strategy, so if it's a mega reuse strategy then I track that strategy closely. If it's a framework strategy I track the documentation part very closely. Any changes architecturally. Any changes, while you're at it, on these use cases. Oh, that's a real problem. Why do we add those use cases? That's just going to chip away at the framework approach. So once I figure out my strategy, then I monitor that strategy, how do they say? Put all your eggs in one basket and monitor that basket carefully." This was echoed by several of the experts on both embedded and nonembedded systems. Person J stated that this is how he controls throughput and latencies. Person D indicated that this control is how he prevents inserting defects. Person E indicated that control is how he controls the cost. Person B points out that if the reuse is not controlled, copies of the reuse

Table 5.18: Ranking of Technical Success Factors by System Type

	Positive		Negative	
Rank	Embedded	Nonembedded	Embedded	Nonembedded
1	Similar Environment or Project	Consistent or Controlled Baseline	Design for Reuse	Searchable Library
2	Standardization	SOA	Searchable Library	Platform Independence
3	Consistent or Controlled Baseline	Test Products/ Autotesting	Trade off	Design for Reuse
4	Design for Reuse	Standardization	Platform Independence	Consistent or Controlled Baseline
5	Experience	Testing	Similar Environment or Project	Documentation
6	Test Products/ Autotesting	Platform Standardization	Experience	Experience
7	Autogen	Searchable Library	Consistent or Controlled Baseline	Standardization
8	Trade off	Encapsulation	Standardization	Portability
9	Testing	Platform Independence	Test Products/ Autotesting	SOA
10	Searchable Library	Design for Reuse	Testing	Maintenance
11	Make fit	Comprehension	Comprehension	Trade off
12	Comprehension	Portability	Portability	Similar Environment or Project
13	Encapsulation	Documentation	Make fit	Test Products/ Autotesting
14	Portability	Similar Environment or Project	Documentation	Testing
15	Platform Standardization	Make fit		Make fit
16	Object Oriented	Experience		Platform Standardization
17	Platform Independence	Parameter Driven		Autogen
18	Documentation	Autotesting		
19	Parameter Driven	Autogen		
20	Maintenance	Trade off		
21	Autotesting	Maintenance		
22	Autotesting	Autotesting		
23	SOA	Object Oriented		

proliferate and the baseline is compromised. Control is closely linked to process. According to the respondents, it is through defined processes that the control is maintained.

Table 5.19: Ranking of Nontechnical Success Factors by System Type

	Positive		Negative	
Rank	Embedded	Nonembedded	Embedded	Nonembeded
1	Process	Process	Process	Estimation
2	Control/ Discipline	Planning	Management commitment	Process
3	Planning	Control/ Discipline	Estimation	Control/ Discipline
4	Management commitment	Culture	Control/ Discipline	Management commitment
5	Culture	Management commitment	Planning	Culture
6	Estimation	Estimation	Culture	Planning

Differences and similarities in technical obstacles Interesting differences emerge when the obstacles to reuse are studied. In this discussion, positive responses were situations in which the obstacle was overcome. Negative responses were situations in which the obstacle caused a reduction in reuse effectiveness.

While both system types identify the same top three obstacles and comment on their negative effect on reuse, the fourth item is different in an important way. Developers of embedded systems mention platform dependence fourth most often as an obstacle, whereas platform dependence is only 6th in mention for nonembedded systems. Further, many in embedded systems do not see platform independence as a goal or even a good idea. Person D says about platform independence, "We're never going to get there and we may not want to. Because you won't get the performance out of the specific device. And the devices are extraordinarily high performance right now." All of the embedded systems experts, but only two of the nonembedded systems experts prefered the idea of platform standardization to platform independence.

Another noticable difference between embedded systems and nonembedded systems developers was the mentioning of the obstacle that the software would be *easier to build from scratch*. While it was sixth in embedded systems, in nonembedded systems it was 14th. Person D stated, "In some cases it took longer to reuse code because of all the issues with it, and it would have been easier to just do it from scratch." According to person K,

"Sometimes it costs the same, and it's an inferior solution to what it would have been if you had just done it from scratch." Person L said, "When people are presented with a pile of code they think that they have to slog through this and develop their own mental image or mental model of it, they think, I'll just write the stupid thing myself or build it in my own model." According to person L, "I think it is just the fear factor that it's less risk if I schedule to build it myself. Then I feel like I have control of the situation." These were all embedded systems experts.

Obsolescence is a big issue. Person C asks, "So, on the producing side, the challenge is how do you produce a reusable artifact that stays useful? " He continues, "There is a concept that I think you've heard me talk about called bit rot, which is anything that sits on the shelf becomes less useful over time. It's a decay function." From person I: "What can be a problem is if the sensor technology changes radically, then in many cases the existing requirements are not appropriate." Person J: "That stuff that we have to deal with all the time, where we say, okay I've got some obsolescent stuff, I've got to move stuff from one platform to another platform, or from one language, which we consider obsolete, to another language." And Person L said, "Plus we would be bridging technologies across a decade or more and the technology in hardware and software is evolving so rapidly that who would want to reuse something that's 10 years old? In avionics and electronics, who wants to reuse something that was designed 18 to 20 years ago?"

Almost every expert in both types of systems cited a problem with what they call *forking*. Forking occurs when a reuse product is migrated from one system to another, and the new system places that product under its own configuration control and proceeds to modify it. When the reusing system does not report the modifications to the originating system along with rationale for modification and complete descriptions of the modifications, the new application is considered to have "forked." When another project is contemplating reusing the product, it is not possible to know the differences between the first and second versions and why they are different. Other problems are introduced by forking. As Person K describes the problem, "The problem there is those people often will run into problems

that were in the code, but they've since kind of forked, they pretty much just copied from somebody else, they don't have the benefit of getting the fixes for most of the problems that were put in the code, they also don't get advances when you update the design and add more features to the design and implementation, they lose those as well."

Table 5.20: Ranking of Technical Obstacles by System Type

	Positive		Negative	
Rank	Embedded	Nonembedded	Embedded	Nonembedded
1	Understanding	Modification	Understanding	Modification
2	Modification	Performance Goals	Modification	Understanding
3	Complexity	Maintenance	Fit	Fit
4	Fit	Certification Process	Platform Dependence	Difficulty - real or perceived
5	Platform Dependence	Understanding	Difficulty - real or perceived	Maintenance
6	Lack of Metrics	Complexity	Easier to build from scratch	Platform Dependence
7	Obsolescence, age	Lack of Metrics	Existing Defects	Unintended Consequences
8	Performance Goals	Fit	Missed Opportunities	Complexity
9	Lack of trust	Obsolescence, age	Lack of Documentation	Loss of flexibility or freedom
10	Existing Defects	Lack of Documentation	Lack of Metrics	Forking
11	Forking		Obsolescence, age	Missed Opportunities
12			Loss of flexibility or freedom	Obsolescence, age
13			Performance Goals	Performance Goals
14			Forking	Easier to build from scratch
15			Lack of trust	Lack of Metrics
16			Unintended Consequences	Existing Defects
17			Certification Process	Lack of Documentation
18			Complexity	Certification Process
19			Maintenance	Insert defects

Differences and similarities in nontechnical obstacles Table 5.21 ranks nontechnical obstacles. There are more similarities than differences in the nontechnical obstacles. *Culture* was mentioned most often by experts in both types of systems. According to Person L, "In my mind the biggest obstacle to software reuse is more cultural and people driven, not technical." The biggest cultural obstacle mentioned was the "not invented here" attitude. This was cited by all but one respondent as a major obstacle to reuse. According to Person E, "developers don't trust work other people have done, especially if the other people developed it for a different purpose. So the level of concern about not invented here grows probably exponentially as it gets further and further from the developers."

Individualism was another obstacle the types of systems shared. This is where the individual developers prefer to build their own products rather than reuse. Person G explains, "But when we have programs that are staffed by younger software engineers, they really view everything that previously exists as having been old-fashioned. And so regardless of how good it was before, they'd rather write it again. They'd rather write it anew. If they look at it and see one thing that they don't like, they're willing to pulse out the whole thing." According to person K, "So many people, it's just their mentality. So many people, they just don't want to reuse, they think they can do a better job even though they know that they're going to spend a similar amount of time doing it. So it's the human factor. Somebody saying I'd rather redo it myself than use someone else's Because it isn't exactly right, I think I can do a better job."

One of the surprising findings in the comments from experts in both types of systems was the impact *contracts* and the way they are written are having on reuse. Person A said, "Some of it (lack of reuse) may also come from the way the contract is written, it says you will do such and such and it doesn't really enable reuse." According to Person I, "One of the problems we have is getting the government to spend the time and money to really do an item by item evaluation of the existing requirements to see if they are truly applicable for the new technology. When they don't do that, we tend to run into some problems." While the DoD Software Reuse Initiative promotes reuse, Person K said, "I'll

be honest with you, one of the biggest hurdles of reuse is really our customers. They tend to put clauses and constraints on the reuse between their programs and other programs. They will often require customer approval. That often kills a lot of reuse from the gate, unfortunately." He goes on to say that it is easier to develop the products from scratch than to get the approval from those customers, and this is costing our government overall a lot of money. This is supported by Person M: "When we talk about our top-level product lines, those are funded by completely separate types of money, budgets, contracts and we can't mix those. If we have created a model under one contract we would not be legally able to use that model in another. So a lot of times it's driven that way. I would say that the major obstacle goes back to that contractual restriction of not being able to use them across major programs."

Finally, *commitment* was cited as a nontechnical obstacle to successful reuse. Often, while management requires the projects to "do reuse," they are unwilling to commit resources, such as time, money, and staff, to creating an environment that encourages reuse. The result is that, as person D pointed out, the requirement to reuse "becomes a tax," that is, an added burden to the staff.

Nontechnical obstacles are serious impediments to successful reuse, but they cannot be overcome by technical staff. It would require efforts on the part of management to overcome these obstacles.

Table 5.21: Ranking of Nontechnical Obstacles by System Type

	Positive		Negative	
Rank	Embedded	Nonembedded	Embedded	Nonembedded
1	Commitment		Culture	Culture
2	Culture		Contracts	Individualism
3			Individualism	Contracts
4			Commitment	Commitment

5.3 Threats to validity

Because semistructured interviews are based on qualitative opinion rather than numerical data, we use the Maxwell threats to validity tailored to qualitative research [90]. These threats fall under the catagories of Descriptive validity, Interpretive Validity, Theoretical Validity, Generalizability, and Evaluative Validity .

Descriptive Validity There is a concern about the factual accuracy of the account of the subjects. The account could be colored by the subjects' opinions or points of view. There may have been enough elapsed time that the subject may not remember as clearly as if he had been interviewed immediately after the event. This threat had to be accepted.

There is also a concern that the interviewer may be mistaken in what the subject actually said. This concern is addressed by having recorded interviews and reviewing the transcripts against the recordings, and by having the respondents review the transcripts.

As mentioned earlier, the differences in definitions of approaches, and to some extent, the artifacts, could have led to overcounting and undercounting.

One difficulty for those wishing to replicate this experiment is having access to many subject matter experts in this many different companies and working on this many different kinds of projects.

Interpretive Validity There is a concern that even if the account of the events is correct, its meaning could be misinterpreted. First, there could be basic differences in the way important terms are used, such as "model" and "component." There could be cultural overloading of words and phrases that cause the interviewer to understand the events differently from how the subject meant them, the "emic" interpretation vs the "etic" interpretation. Often emic interpretations are heavily influenced by the culture , its customs and beliefs. Thus, the subject's analysis of the events could be colored by beliefs widely held in his organization or location. On the other hand, the interviewer could be making an "etic" interpretation, that is, interpreting the events in a cross-cultural

mannerand not understanding the impact of the local or organizational culture on the reporting of the event. In our case the interviewer, as a former employee, understands the termnology as well as the culture.

In addition, the email that asked for participation defined the key terms used by the interviewer. If there was ambiguity in a term used, the subject was asked for a definition. While this did not remove the ambiguity introduced by culture and beliefs, it helped with the linguistically introduced ambiguity. The definition the subject used was then applied to the subjects' reports on their events.

Theoretical Validity "Theoretical validity goes beyond concrete description and interpretation and explicitly addresses the theoretical constructions that the researcher brings to, or develops during, the study [90]." We have not identified theoretical threats to validity.

Generalizability "Generalizability refers to the extent to which one can extend the account of a particular situation or population to other persons, times, or settings than those directly studied [90] p.293." The main threat to validity here is whether this research can be extended to other industries, or to organizations with different processes, procedures and training. This threat is mitigated to some extent by the variety of sites and projects studied. In addition, there is doubt as to whether these results can be extended to smaller projects. However, the generalizability of a study where the judgment method is used to select the candidates is limited. Generalizability was not a goal of this study, rather, it was to understand the reasons for differences.

One major threat to validity is the size of the projects studied. All of these projects are extremely large. The embedded systems are very large government acquisitions with multiple subsystems and highly refined performance requirements. Most of the nonembedded systems involve big data, with high requirements for processing and throughput. Even the the smaller Independent Research and Development (IRAD) are intended to be migrated to very large systems. Many of the obstacles for these large systems may not exist for smaller or less controlled systems.

Evaluative Validity Evaluative Validityis the question as to whether the researchers were able to describe and understand the events without being evaluative or judgmental. It involves the researchers removing their own biases and grouping and decomposing reports of the events. This threat is compounded by the fact that the respondents were being asked to give their own opinions on the topics, that is, they were asked to be evaluative about the subject. To alleviate this threat, the researcher took great pains to ask questions that were neutral, and to limit responses to indications that the respondent had been heard (without further comment).

5.4 Conclusion, Lessons Learned and Future Work

One of the original questions was, Is reuse truly beneficial? While the interviews do indicate that reuse is often of great benefit, they also indicate that reuse is not always a good solution to a problem, and that it comes with cost. Reuse requires research into available products and thorough analysis of the candidates selected. Failure to understand the design, interfaces, performance and other aspects of the reuse can cause the outcomes to be poor. While most respondents cited design for reuse as an important factor in making reuse work, there is added cost to developing for reuse. The project managers need to determine whether the artifacts will be reused enough to recuperate the added cost before the artifact becomes obsolete. Most of the respondents discussed design for reuse and code together - designing the code to be reused. However, as person L pointed out, "code is especially difficult to understand, because it is not visual, and sometimes it is hard to follow."

Most respondents indicated that if a candidate artifact is not well documented, if any modifications need to be made, the artifact is not reusable. This comment was especially made in reference to code. Person L points out that "models are graphical, and therefore much easier to understand. Models become their own documentation."

Locating potential reuse products was cited as a difficulty by nearly every respondent, both embedded systems and nonembedded systems. Most had tried libraries of some sort, but the libraries had significant problems. They were not easy to use, they did not carry documentation, they were not often populated by projects who had developed similar products.

The way contracts are written was another issue that inhibits reuse. While on the one hand, the government is mandating reuse through the software reuse initiative, many of the contracts have clauses that make porting artifacts from project to project too difficult. Rather than go through the process, the developers consider it easier to just redevelop. Management needs to address reuse and the way contracts are written to overcome this issue.

While all of the respondents indicated that they are reusing code and requirements, those were not the artifacts that provided the highest return on investment. Every respondent except one placed code and requirements somewhere between the middle and the bottom of the list. For ten respondents, architecture and design artifacts offered the greatest return, for four, test products. Interestingly, the four who ranked test products highest were all from embedded systems.

The semistructured interviews did indicate that embedded systems and nonembedded systems, while in many ways similar, are different in important ways. Embedded systems experts endorsed platform standardization but did not favor platform independence. Nonembedded systems experts favored platform independence. The ability and need to optimize to a platform is important to embedded systems. Most of the nonembedded systems are in virtual environments and need to be platform neutral. One could reasonably claim that the virtual environment is their platform, as Person B suggested. Embedded systems reuse certain artifacts, in particular, code and requirements, because they were already developed for the platform. Embedded systems experts found that a similar envorinment was important, nonembedded systems experts found it much less important. This may be because the nonembedded systems were being built in a web-based environment,

so the similar environment was already established. Most of the embedded systems experts felt that it was easier to build products from scratch, whereas the nonembedded systems experts felt less so.

Having identified these differences, it is important to understand why the difference? The difference in model reuse is particularly interesting. The embedded systems were, for the most part, much older systems. The first instantiation of many of them predated modeling tools. Two of the embedded systems respondents indicated that if they were to start today, they would use architecture models and design models. However, migrating and reengineering at this point is too expensive, the systems are working, and the technology insertion is achievable. Over time, these systems may phase out and more embedded systems will use more models up front. In fact, three of the nonembedded systems had been reengineered and were using SOA, and two of the embedded systems had been reengineered into product lines with a reference architecture as the center. All of the reengineered systems were reusing models extensively, the embedded systems that had been reengineered were also reusing many components.

The difference involving platform independence may be more intractable. Especially in core software, that is, software written directly to the processor as is an FPGA or an ASIC, the software is optimized to the platform. Once it is coded, it is very hard to rehost. However, as performance modeling tools and design modeling tools have been used more often, each with autogenerating capabilities, the platform specifics can become attributes in the model. Platform standardization allows the models to carry the specifics of the different platforms. This allows the models to be reused and the code to be specifically generated for each instantiation. Over time, we see code being regenerated for each use, with the reused products migrating away from code and requirements and to the models. This will enable more reuse and more successful reuse, as was the experience of person G.

The question of a similar environment is somewhat related to the platform independence. Embedded systems are difficult to port to different environments. It may be that in fact, the same is true for nonembedded systems, but the virtual environment is in fact

the same even if the application does very different things. However, changing that environment is problematic. We found this to be the case when Person I discussed migrating a ground based nonembedded algorithm into an airborn embedded algorithm. "I think platform independence can cause a problem. Let me give you an example. For the ground data processing systems, because we standardized on X 86, and VMware and whatever else, we have these highly tuned algorithms that can be moved from X 86 system to X 86 system. But then they say to us, those algorithms are great. We want him to run on an embedded system in an airborne platform. I say, well that's great. Are you running X 86 systems in your embedded? Well, we've got these GPU's that we are using to save size, power, and weight, and blah blah blah. All of the sudden now it's a totally different platform. To make these algorithms work on the embedded platform is a huge cost."

Given these differences, these interviews suggest that it is unwise to assume embedded systems are the same as nonembedded systems, or that they should be handled in the same way. For now, they should have different sets of processes and different metrics. Some research is being done on developing metrics already [105], but more needs to be done.

However, these interviews also suggest that the two may be converging. Many of the older systems have been modified to the point where they have become brittle. They will have to be reengineered or phased out. If the aforementioned projects are an indication, the embedded systems will move to product lines, and the nonembedded to SOA. A closer look at the reengineered projects shows a great deal of similarity in the SOA and product line solutions. Both use reference architectures, based on models. The services mentioned in nonembedded systems differ from the components in the embedded systems only in that embedded systems components include the specific hardware. Both embedded systems and nonembedded systems developers are using models to autogenerate code. So, while code has been the most frequently reused artifact in the past, with the increased use of models with autogeneration, it may become less and less common in the future. In fact, Person N said, "Essentially a coder is going to, in the future, be a software technician, rather than a

software engineer. My hope is that we don't need software coders any more, because we'll develop robust graphical software tools that will allow us to unambiguously auto generate the code."

Lessons Learned These interviews provided a number of lessons.

1. Reuse is not always a good solution to affordability. Both embedded systems experts and nonembedded systems experts cited examples where reuse was not beneficial.

2. Work needs to be done up front to identify and understand reusable products. As several of the experts pointed out, the decisions made in early stages of development impact the success of the reuse for the rest of the life cycle.

3. There are subtle but important differences between embedded and nonembedded systems. These differences tend to be related to reuse of code.

4. if management is going to mandate reuse, they need to provide the developers with a searchable library of well developed products and complete documentation.

5. If reuse is to be feasible, contracts need to be written to accommodate reuse.

6. While most frequently reused, code offers the lowest return on investment.

Future Work There appear to be benefits to reuse in each development approach. Many of these benefits in development approach appear to be complementary. For example, model based development overcomes the problem of platform dependence, while component based development enables reuse of unmodified code to specific platforms. By putting specifics of different platforms into a model and allowing the developer to select the platform, the model can help the developer select the best component already developed for his requirements. Then the developer would only have to develop artifacts for the system that have not yet been developed for the platform. A product line could collect both models and components, allowing a developer to quickly assemble the needed artifacts for the

system. A topic for further study would be to identify the strengths and weaknesses of each development approach and use this information to formulate an approach that takes advantage of the strengths and minimizes the impacts of the weaknesses. This is discussed in the next chapter.

Chapter 6

Creating a New Framework to Enable Reuse

In our research, we have sought to identify what reuse strategies or combinations of strategies are most (and least) effective for various types of projects, and which artifacts are most effectively reused. We have identified success factors for reuse effectiveness and obstacles. We performed a review of existing literature for clues to reuse practice and impacts. We conducted a survey of software practitioners to identify the development approaches they actually use and the artifacts they reuse. We performed a series of semistructured interviews to gain insights into the factors that enable success and the obstacles they encounter. We synthesized the information obtained in these studies to posit a framework for reuse that maximizes the success factors and neutralizes the obstacles.

6.1 Summary of Existing Literature

To begin our research into reuse practices in embedded and nonembedded systems, we looked at extant literature [7]. While we found many, many articles discussing reuse, some in embedded systems and some in nonembedded systems, we found none that directly compared the two types of systems. We found many articles discussing reuse in different development types, but we found none comparing reuse in different development types.

We found seventeen empirical studies covering reuse in embedded software. Seventeen other studies involved reuse in nonembedded software. Nine studies covered development strategies in both embedded and nonembedded systems, however, these studies did not

compare embedded systems to nonembedded systems. Seven types of empirical studies were represented in our review, three were of the more rigorous type (case study, quasi-experiment, survey), two were a less rigorous type (review of practice and meta-analysis) and two were the least rigorous (expert opinion and experience report). In these catagories there were five studies of embedded systems, nine studies of nonembedded systems and three of both embedded and nonembedded systems were of the most rigorous types. No studies of embedded systems, one of nonembedded systems and two studies of both embedded and nonembedded systems were of the less rigorous types. Twelve studies of embedded systems, seven studies of nonembedded systems and four studies of both embedded and nonembedded systems were of the least rigorous type of study.

Of the aforementioned studies, only those we refer to as rigorous or less rigorous offered quantitative data. Qualitative data was available for product line (two studies), and model based development (three studies). There were no quantitative data for computer based development, and the studies that did not specify their development approach also did not include quantitative data. All studies did offer qualitative data. This classification was used to identify and compare outcomes based on the metrics discussed in the papers.

We found that when we searched the papers to determine how success was measured, the metrics fell under five basic categories: size, reuse levels, quality, effort, performance and programmatic (such as staff, institutionalized process, or schedule). We catalogued the specific metrics used under these categories for comparison. That comparison showed that while both embedded systems and nonembedded systems reported size, reuse levels, quality and effort measures, their lower-level metrics were different (for example, in quality, both reported on defect density, while only embedded systems reported on reliability, and only nonembedded systems reported on severity, and only studies of both embedded and nonembedded systems reported on source of error). Only embedded systems reported on performance measures, and only studies of both embedded and nonembedded systems reported on programmatic measures. Already, we were noticing differences between embedded and nonembedded systems.

We next turned to outcomes. Given the diverse ways in which various system attributes and reuse variables were measured, as well as the lack of effect size in most studies, we could not perform a metaanalysis that could quantitatively assess and compare similarities and differences between outcomes. In an effort to quantitify and analyze the data, we include as "characteristics of events" the concepts of "better, worse or mixed." and assigned an ordinal measure to outcomes. A relatively large number of papers only reported very high level results rather than results for specific attributes of reuse outcomes. These are scored similarly, classified under "general." In the narrative that follows, we report in parentheses the embedded systems normalized scores followed by the normalized scores for nonembedded systems.

Once the scores were normalized to account for differences in the numbers of projects reported on, and the normalized scores were compared, and we noted that in every case, nonembedded systems projects reported higher positive reuse experience than did embedded systems projects. On the other hand, more embedded systems projects reported negative reuse experience than did nonembedded systems. In particular, the measure of quality was reported as worse more than seven times as often as for nonembedded systems (35% to 6%), and effort was reported as negative more than five times as often for embedded systems as for nonembedded systems (28% to 5%). The overall experience for embedded systems projects was reported negative at nearly 5 times the rate of nonembedded systems (14% to 3%). This is an important difference between embedded and nonembedded systems.

As we moved into the development approaches, we found that for product line, the reported measures were the same for embedded systems and nonembedded systems, except for the reuse levels (64% vs 88% positive, 36% positive vs 12% negative) and general reuse (81% vs 19% positive, 19% vs 0% negative). There was a large difference between embedded and nonembedded systems in model based development in all metrics. While reuse level was higher for embedded systems in model based development (77% to 65%), effort, quality and general results were all worse for embedded systems (effort 19% vs 53% positive, 44%

vs 11% negative; quality 15% vs 33% positive, 33% vs 8% negative; general 22% vs 53% positive, 11% vs 7% negative, 67% vs 40% mixed). In component based development, while both embedded and nonembedded systems reported higher reuse levels and a general positive reuse experience, there were again important differences with the other measures, with embedded systems reporting less positive and more negative results (effort 33% vs 67% positive, 33% vs 0% negative; quality 0% vs 100% positive, 75% vs 0% negative). We note that there were too few observations in component based development projects to draw conclusions about these numbers. Results were interestingly mixed when development approach was not identified: while nonembedded systems projects reported worse results in terms of reuse levels (0% vs 20% positive, 50% vs 80% negative), nonembedded systems projects reported much better results in terms of effort and quality (effort 67% vs 100% positive; quality 0% vs 100% positive, 75% vs 0% negative). In both embedded and nonembedded systems projects 100% reported positive experiences.

We tested the hypotheses for significance using Chi-Square. We found a significant difference between embedded and nonembedded systems when measuring effort in model based development and measuring quality when the approach is not specified (p-values of .0292 and .0005 respectively). This tells us that:

1. Savings in effort are less likely to be successful for embedded systems when a model based approach to reuse is employed.

2. Quality improvements are less likely to be realized in embedded systems (in studies that did not report on the development approach used).

We searched the analysis portion of the appropriate papers to see if we could identify reasons for these results. Since the studies were not comparing development methods, the closest explanation we found was in the DARPA project [118], wherein the researcher explained, "Our implementation was not carried out through widespread modeling, and there are a few reasons for this:

- the diverse team of experts in robotics, computer vision, software, and control were not all familiar with software modeling techniques;
- the operating environment of real-time behaviors required many components to run on a real-time operating system with limited tool support;
- the behavior of many components is best specified using general-purpose techniques, especially the advanced control algorithms used."

We concluded that, indeed, there were differences between reuse in embedded and nonembedded systems. While inexperience with the modeling tools may account for some of the negative reuse experience, we were not convinced that this inexperience sufficiently explained the significant differences. It seemed that there had to be more reasons for these differences than inexperience with tools. In order to further understand these reasons, we decided to perform a survey of practitioners. This survey would identify actual reuse practices in an aerospace environment and inquire into the reuse experiences.

6.2 Summary of The Survey

The review of existing empirical studies about reuse comparing the reuse outcomes for embedded systems against nonembedded systems using different development approaches revealed that reuse in embedded systems leads to significantly less positive outcomes than in nonembedded systems when the development approach is model based engineering, and that, overall, reuse in embedded systems is less successful than reuse in nonembedded systems. There were also some indications that some of the difference in outcomes could be related to the artifacts reused, but few of the empirical studies focused on artifacts and their impact on reuse. In fact, while not explicitly stated, it appeared that most of the projects studied in the literature were reusing code. When other artifacts were considered, their impact on the success of the reuse was not studied.

Our search of existing literature left several questions that we felt needed to be answered in order to understand and implement successful reuse. First was whether industry practitioners share the same reuse experience as the research for embedded and nonembedded systems. Second is the question of what artifacts can be reused for successful outcomes. Third is the whether there is a difference between embedded systems and nonembedded systems in outcomes.

Because our questions were of the form described by Yin [**?**], we decided to use a survey to pursue these questions. To answer these questions, we conducted a survey of aerospace professionals. Eighty-two subjects responded to the questionnaire. Since four did not identify their system type, we discarded their answers, and proceeded with the answers from 78 respondents. Of the 78 subjects, 41 reported on embedded systems, while 37 reported on reuse in nonembedded systems.

To answer our first question, we asked about development approaches used. 33 projects used only one development approach, excluding four that did not report using any development approach. Of these, 17 were embedded systems, 16 were nonembedded systems. The rest used a combination of approaches. While nonembedded systems were more likely to use ad hoc, COTS/GOTS, and even no approach, embedded systems were more likely to use component based, ad hoc and product line development approaches. This was also true when considering combinations of approaches. There were 22 different combinations of approaches, and no one combination approach was used by more than four projects. Most of the composite approaches were used by only one or two projects. The most commonly used development approach for embedded systems included component based development, while nonembedded systems included Ad Hoc reuse and a Product Line approach most frequently. Embedded systems projects used component based, model based and product line approaches more frequently than nonembedded systems, while nonembedded systems used Ad Hoc reuse and a COTS/GOTS approach to reuse more than embedded systems. Interestingly, the most commonly used approaches in embedded system projects are not the most commonly used in nonembedded system projects and vice versa.

To answer our second question, we asked about artifacts reused. We found that reuse levels for all artifacts is higher in embedded systems projects than nonembedded ones. Sometimes the difference is small (reuse of models, drawings), but often reuse levels in embedded systems are at least 10% higher higher than in nonembedded systems (e.g. requirements, architecture, use cases, hardware, test products and test clusters). Again, we found a difference between embedded and nonembedded systems.

We asked about outcomes from reuse. Outcomes would include savings in labor costs, fewer defects, less time for testing and fewer items to test. Because many of the projects had not yet reached implement, integration, and/or test, many of the respondents were not able to provide information on outcomes for defects (usually found in test) or test time. Box plots of the responses to the outcomes questions showed us the medians and spreads of the answers comparing embedded systems to nonembedded systems. We were surprised to see that the medians of each of the outcomes were equal for embedded and nonembedded systems, although the spreads were very different. The medians also varied by outcome type, with labor savings being the highest (10-20% savings), and defects and items tested showing no savings. Nonembedded systems showed a greater spread than embedded systems for labor and defects. There were, however, no outliers in nonembedded systems in any of the outcomes, whereas test time reduction and test item reduction showed outliers in both directions for nonembedded systems.

Next, we asked about whether risk was mitigated or introduced by reuse. This yes or no question indicated little difference between respondents working on embedded systems from nonembedded systems when asked about whether reuse reduced risk. 31 of the subjects reporting on embedded systems projects thought reuse reduced risk, nine did not, and one did not respond. Conversely, 29 of the subjects reporting on nonembedded system projects thought reuse reduced risk, while eight thought it did not.

We tested the hypothesis via MANOVA and found that there was no significant difference ($p<-.05$) between embedded and nonembedded systems in the development approaches selected. We could not reject the null hypothesis that the development approaches did not

differ between embedded and nonembedded systems, although use of component based development was close to significant at p=.055. However, in artifacts used, we did find significant differences. Requirements (.040), code (.033), models (.009), drawings (.001) and already tested clusters (.049) were all significantly more likely to be reused by embedded systems to the .05 level, and architecture (.051) was close to significant. Based on our observations from the box plots, we did not perceive a value in comparing outcomes with MANOVA.

Because of the findings about artifacts, we decided we needed to run PCA using AHP. Our question was whether the reuse of the artifacts more commonly used by embedded systems were used together, and whether system type or development approach led to greater use of certain artifacts.

When we perform PCA on all variable responses, we observe several survey response relationships which are unique. Selection of a heritage/ad hoc reuse approach results in less use of models as reuse artifacts, and greater use of tested clusters. The selection of models as reuse artifacts indicated less heritage/ad hoc reuse. However, more tested clusters indicates surprising less heritage/ad hoc reuse. We confirmed that a COTG/GOTS reuse approach leads to less reuse of use cases. Two interesting relationships are between system type, architecture and hardware in that a three-way relationship in which an embedded system type leads to less architecture reuse and more hardware reuse. We conclude in general that reuse results in a reduction in test time. Our PCA results, however, do not allow us to correlate test time reduction with any particular reuse approach or artifact.

We found that system type does contribute to variance such that reuse approach, reuse artifacts and outcomes are influenced by system type. I.e., their variance in terms of inter-relationships is affected by which system type is used.

A summary of the significant PCA findings from survey response relationships are listed below:

- A reduction of items to be tested is not associated with any particular system type, reuse artifact or reuse approach when outcomes are excluded

- A reduction of test time is not associated with any particular system type, reuse artifact or reuse approach
- Experience with embedded systems tends to avoid a COTS/GOTS as a reuse approach
- More often than not, development approach and reuse artifact vary together but independent of both system type and outcome
- Embedded systems reuse fewer models and use cases, but more tested clusters when reuse approaches when analyzed without reuse approaches

In general, our PCA confirmed that the selection of embedded vs non-embedded system type does not always relate to outcomes such as an increase or reduction in risk, or an increase or reduction in test time. Some test pairs showed an independence in the relationship between reuse approaches and reuse artifacts, from system type. A notable exception is that when reuse artifacts are excluded, risk is reduced through a non-embedded system type and through the selection of a heritage/ad hoc reuse approach, in isolation of responses for reuse artifacts.

From the survey, we found that there are differences between embedded systems and nonembedded systems in development approach and reuse artifacts, although not in outcomes, but we did not know why. For example, are requirements driving reuse of use cases, architecture, code and already tested clusters? Or is the selection of certain reused components driving these artifacts including requirements? What is driving the selection of development approach and reuse artifacts? To answer these questions, we decided to interview experts in the field.

Finally, we looked at the free-form answers in our survey to see if we could identify reasons for the similarities and differences we found above. Developers of embedded systems cited benefits from using component based reuse: not having to reengineer or rewrite the test sets; the technical solution was clear and already deployed; pedigree of the products; reuse of a proven methodology; developer confidence in the products and not touching

common code or infrastructure. Reasons for a model-based approach (for both embedded and nonembedded systems developers) include that models had been used in other projects. Reasons in favor of product line reuse in nonembedded systems included risk reduction, a good match of the reused software with the new project, and the development of reusable expertise. No reasons were provided for the benefits in reuse of COTS/GOTS.

There were also differences observed in the artifacts used between developers of embedded and nonembedded systems. Both embedded and nonembedded systems practitioners mentioned software, code, components and test. However, while hardware was a major consideration for embedded systems, it was not mentioned by nonembedded systems.

Presumably, reuse in embedded systems is facilitated when code does not have to be adapted to new hardware. This is cited as a reason for success and a reason for reuse failure when hardware does change. Well known and documented circuits, stable hardware, and running the software on known platforms, not having to reengineer or rewrite test sets reduced risk. In cases where code reuse is feasible for embedded systems, it allows for higher efficiency, lower risk. Risk was also reduced by reuse of code because it had been tested multiple times and the fact that a large percentage of the code was already integrated, and because the of the use of proven equipment, designs and requirements.

Unsuccessful reuse in embedded systems was explained by not being able to use or "fit" the reuse artifact: architecture mismatch with third party code, code or reuse plan mismatch, lack of availability or obsolete test products, overly complex software artifacts, and problematic legacy software.

The close coupling between hardware platforms and software is cited as a reason for successful reuse when the platform is unchanged. The technical solution was proven in an environment where performance and timing are critical. The fact that the hardware and software had proven themselves and the likelihood of introducing new defects was reduced was an important consideration. Developers knew that these products worked well together. While these were also cited by developers of nonembedded systems, embedded systems developers clearly relied on the past success of their components for new efforts.

In nonembedded systems, respondent comments related to successful reuse cited good documentation, a good fit of the reused software to the new use, and availability of product-line artifacts. The platform was not mentioned.

By contrast, mismatches of reused software with needs for the new project, poor quality of code or design, poor documentation of reusable artifacts, and complexity of the code base were cited as reasons for increased cost or failure. Undocumented or latent problems were cited as reasons for much higher testing efforts.

6.3 Summary of Results from Structured Interviews

To gain understanding of the findings from our review of existing literature and our survey of systems practitioners, we decided to conduct a series of semistructured interviews of recognized fellows and distinguished staff. Of the fourteen interviews, seven experts were primarily involved in embedded systems, six were primarily involved in nonembedded systems, and one was involved in both embedded and nonembedded systems. We asked them about the development approaches they used, the artifacts they reused, the level of reuse, obstacles they encountered, success factors, impats of nonfunctional requirements, and the need for reusing the hardware along with the software. Then we compared the answers between embedded systems experts and nonembedded systems to identify commonalities and differences. We also examined their free-form comments for clues as to why these commonalities and differences existed and for insight into reasons for reuse success and failure.

Results of direct answers to questions

First we asked about development approach. In embedded systems, four experts used ad hoc reuse, three used model based development, one used component based software development, three used product line and one used ontology. One used what he called mega reuse, that is, he took the software intact and did not modify it, and one described his development approach as a framework. This number adds up to more than the number of

subjects because three used a combination of ad hoc reuse and model based development, two used a combination of ad hoc reuse and product line, one used a combination of model based development and ontology. In nonembedded systems, four used ad hocreuse, one used component based software development, one used model based development, one used the mega reuse discussed above, one used a framework and one described his development approach as SOA (probably most like model based). This differed from the total number because one used a combination of ad hoc reuse and model based development and one used a combination of ad hoc reuse and model based development. Other than product line, which was reported used 21% of the time in embedded systems and not at all in nonembedded systems, and ad hoc, which was used 44% of the time in nonembedded systems and only 29% in embedded systems, the development approaches did not differ substantially between embedded and nonembedded systems.

Our second question was about artifacts used. every subject reused code, requirements and design. However, we begin to see differences between embedded and nonembedded systems in other artifacts. For example, nonembedded systems experts were far more likely to report reusing architecture (71%) than embedded systems experts (25%). 63% of embedded systems experts reported reusing components, compared to zero for nonembedded systems. No embedded systems experts reported use of SOA, but 43% of nonembedded systems experts did. Test Products reported use by 75% of embedded systems experts, compared with only 57% of nonembedded systems experts No reuse of design products was reported by embedded systems experts, while 71% of nonembedded systems experts did use them. The rest of the artifacts reported similar levels of reuse.

We asked about the obstacles to successful reuse. We found that, of the 24 obstacles named by the experts, eight were mentioned by at least 20% more embedded systems experts than nonembedded systems experts, and seven were mentioned by at least 20% more nonembedded systems experts than embedded systems experts. Only nine had less than a 10% difference. We concluded that the obstacles to reuse were different between embedded and nonembedded systems.

Results of coded freeform comments

At this point, the responses were coded in an excel spreadsheet. Each statement was classified as coming from a respondent specializing in embedded systems or nonembedded systems, and then coded as being positive, negative or informative (P, N, I). Informational was where the respondent was describing a concept without making a valuation of the concept. Many of these statements provide insight into the reasons for some of the observations.

Development Approach

We noticed that, while most embedded systems experts used *ad hoc* as their development approach, there were only six positive comments about that approach and 11 negative. Conversely, while three were using a *model based* approach, positive statements about model based development exceeded negative comments by 16 to one. Embedded systems experts also had positive comments about *product line* 10 times against only one negative comment. It is clear that many embedded systems experts were not using their preferred development approach. Reasons for this ranged from knowing the code and knowing the architecture to the fact that the projects they were working on had been around for many years and were using the old code. If they were starting a new project, or if the existing project were to be reengineered, they would prefer to use a model based approach. A surprising difference in the nonembedded experts is that the development positively mentioned most often was*ad hoc* (8 times), although it was also mentioned in a negative way nine times. It appears that it is easier to import scavanged code into new systems for nonembedded systems, whereas embedded systems require more tailoring. Since several nonembedded systems experts mentioned SOA, whereas no embedded systems experts did so, perhaps the code nonembedded systems experts are importing are more easily treated as services, which are not optimized to a platform.

Artifacts

Reuse of components and models received the most positive comments from embedded systems experts (70), with architecture second (60), code third (57), design fourth (51)

and requirements fifth (42). Hardware, design patterns, and interfaces were mentioned positively 24, 23 and 22 times respectively, test products 18 times, nonfunctional requirements 13 times, documentation 11 times,algorithms 8 times, test clusters and concepts each 5 times, and services once. Service level agreements and data products were not mentioned. The most negative mentions by embedded systems experts were code (42 times) and requirements (24 times). Nonfunctional requirements and design each had negative comments 14 times, followed by architecture and hardware with 13, interfaces with 11, components and models (9each), documentation (8), design patterns (3), algorithms, and test products (2 each), COTS and test products (1 each). Services and service level agreements were not mentioned by embedded developers. Informational statements about code were made most often (20 times), followed by hardware (17 times), components and models (13 each), architecture (9), requirements (8), interfaces and data products (5), non- functional requirements (4), design and test products (2), and concepts, COTS, algorithms, test clusters and documentation each got one informational comment. Service level agreements, services and design patterns were not the subject of any informational comments from embedded systems developers.

While embedded systems experts cited code, design and requirements as the artifacts used most often, these were not the artifacts they talked about the most. Code and requirements were artifacts from legacy systems, and the requirements had already been implemented in code. Since the requirements were essentially unchanged, the rest of the assets could be reused. Interestingly, one expert stated that, "I don't know that I've ever been on a program that just said let me reuse requirements or design. Typically it started a code reuse, and flexibility of that code applicability of that code being utilized in full or some modification to it for your project, then the documentation, and the design and the requirements, come with it." So the embedded systems experts start with an understanding of existing code and import the artifacts they need along with that code to develop their systems. However, as one pointed out, as autogen becomes more robust, the need to reuse code will decline.

Negative comments by embedded systems experts about requirements, design and especially code offer a great deal of insight as to why code reuse often fails: people do not write reusable code because they're in a hurry to get designs out; they write in low level languages like assembly code; the lack of doing a high level preparation; the lack of documenting the requirements or making sure the requirements are comprehensive, complete, accurate; and lack of testing correctly. Much older code was not subjected to the correct development processes, and is brittle. Many indicated that If the code needs modification, it is almost never effective to reuse it. One pointed out that it is hard to build a mental image from a bunch of lines of code, and if that's what you are given, it's easier to build it from scratch. Without solid documentation, including graphical representations, it is easier to create the module than to try to understand the candidate reuse artifact.

In the embedded systems, the reuse of models was particularly interesting. While all of the respondents use models, embedded systems experts found the performance models and simulations the most useful. In fact, some regenerated their code anew with each instantiation from the performance models. In many cases, their design models and their performance models were one and the same. All of the embedded systems experts mentioned using performance models to ensure their system would meet performance requirements before moving to implementation. These models were reused from instantiation to instantiation, with the performance models modified to reflect new performance requirements. There were also a few negative comments about model reuse. The challenge is translating the model into code for the core processor. There is also the question of whether the cost of developing a model is justified.

Unlike embedded systems experts, nonembedded systems experts positively mention reuse of architecture most. They pointed out that architecture, particularly a reference architecture, enhances reuse success because architectures abstract away many of the variabilities that affect reuse. Abstracting away the specifics of the implementation allowed the developers to reuse an abstract architecture and add in the particulars during development.

While nonembedded systems experts did make many positive comments about reusing code, they also believed reuse of code introduced problems, especially if the code needed modification. They pointed out that the reused code could force the requirements and architecture and result in a less than optimal solution. In addition, they noted that if the code had to be modified, defects were frequently introduced.

Success Factors

Embedded systems experts named a similar environment or project as the most important technical reuse success factor, followed by standardization and a controlled or common baseline. Lack of design for reuse and a searchable library were most frequently cited as success inhibitors. Nonembedded systems experts cited a controlled or common baseline as the most important success factor, followed by use of SOA and autotesting. While both types of systems considered the baseline as one of the top reuse success factors, their next two top choices differed. We note that standardization (i.e. data standards, platform standards, interface standards) and similarity of project are important to reuse success in an embedded system. This is attributed to the fact that the software is written directly to the processors and must adhere to strict I/O and timing requirements.

Obstacles

Both embedded and nonembedded systems experts point to the same three top technical obstacles to reuse: fit, understanding and modification. Experts of both systems types reported these obstacles as being related to code: whenever the code has to be "broken open," problems arise. It becomes difficult to trace changes through all of the code modules, defects are introduced, the code has to be retested in the system and against the platform. Embedded and nonembedded systems experts also found that platform dependence was another major obstacle to reuse. This is interesting, because only code is specific to the deployed platform.

6.4 Analysis of Results

6.4.1 Differences and Similarities between Embedded and Nonembedded Systems

Differences From the review of literature, we found that there are significant differences between embedded and nonembedded systems in the outcomes for quality in model-based development and general development. We found from the survey that important differences in the development approaches in that embedded systems developers tended to reuse component based development, product line, and model based development most often as a part of their development strategy, where nonembedded systems developers tended to prefer ad hoc and COTS/GOTS development approaches.

It was surprising to find in the survey that embedded systems developers reused all of the artifacts more frequently than did nonembedded systems developers. We found that in embedded systems, requirements, architecture, code, hardware and test products tended to be used together. We wondered whether that is because large portions of a system were imported for reuse together. We also wondered whether advances in technoloty were being missed because of the cost to redevelop certain capabilities and therefore affordability was hampering the technology.

Similarities We found that average outcomes were the same for embedded and nonembedded systems. Both were realizing good improvements in labor, but improvements in quality, testing time and items to be tested were disappointing. These are areas that need attention.

6.4.2 Summary of Benefits and Detriments of Development Approaches

Benefits of component based reuse include not having to reengineer or rewrite the test sets, a clear and deployed (proven) technical solution, pedigree of the products, and developer confidence in the products (R28). Not touching common code or infrastructure was

important to embedded systems developers. Benefits of a model-based approach include that models had been used in other projects that reduce their risk. Benefits of product line reuse included risk reduction, a good match of the reused software with the new project, and, over time, the development of reusable expertise. The benefit to ad hoc development is that code already exists that realizes the requirement(s).

Detriments of component based reuse include aging or obsolescence of existing products, difficult and/or proprietary interfaces to the components (including hardware interfaces, differences in the platform libraries and components that don't exactly match requirements. Detriments of model based reuse include poor fit of models to the new system. Detriments to product line development include limitations on the systems built into the product line. Detriments to ad hoc reuse include an imperfect fit into the system or requirements and uncertainty of their development process and pedigree.

6.4.3 Summary of Benefits and Detriments of Artifacts Reuse

A benefit of code reuse is that it does not have to be adapted to the hardware, i.e. well known and documented circuits, stable hardware, and running the software on known platforms, and not having to reengineer or rewrite test sets, leading to higher productivity and fewer defects. When reuse of code is accompanied by reuse of component interfaces, components, requirements and test procedures, quality is increased and risk is reduced. We note that many reported that a benefit to the reuse of code is that it has already been tested. This would imply that it was not changed, because modifications would require retesting. A benefit to the reuse of requirements and design and is that the solution is already available.

A benefit to the reuse of models is that they are graphical, self documenting and easy to modify for the new system. Models are also independent of the platform to be deployed in the system, and can contain the specifics of multiple platforms, allowing the developer to select the target platform. Many modeling tools have the ability to autogenerate code and documentation.

A benefit to the reuse of simulations is that many are already adapted to the environment in which the system will be deployed (i.e. big data applications, spacecraft and aircraft) and require only a change in parameters to produce the analysis.

Benefits to the reuse of COTS and components is that they are already available and often inexpenisive.

Detriments to the reuse of code include aging and obsolescence. If the code requires modification, it can be more difficult to trace the impacts of those modifications through the code, especially if the code is tightly coupled, resulting in the insertion of defects. Reuse of existing code can dictate the architecture and the requirements to force a suboptimal solution. It can force the selection of outdated hardware. Often, the code can be hard to read, understand, and analyze for fit and function or to identify how difficult it will be to modify. It can be complex. The same detriments apply to the reuse of COTS and components.

Detriments to hardware reuse include the close coupling between hardware platforms and software. Differences in the platform libraries and performance and timing when porting code from one platform to another is another problem.

6.4.4 Summary of Success Factors

From the semistructured interviews, we were able to identify a number of technical success factors:

Planning up front Both the embedded and nonembedded systems experts stated that the time spent in planning the system up front was time well spent. When the developers started developing a system without up front planning, they almost invariably wound up selecting inappropriate products for reuse and needing to correct defects throughout the life cycle. When the planning was done, the products selected were more likely to fit into the new system and realize cost and schedule benefits.

Control (authoritarian) Several of the systems and software experts indicated that when they were employing reuse they had to be even more authoritarian in how the products were handled and how the system was developed than if the project was developing new products. The software architects had to ensure that the developers did not rework the software (i.e. divide one component into two components or change the way they interface). One mentioned that the way that reuse was designed was how he calculated his latencies and througput budgets. When the reuse was tightly controlled, it almost always provided benefit.

Design for Reuse While designing code for reuse is usually 30% more expensive than for a single implementation, selecting code for reuse that was not designed for reuse would usually lead to increased cost and schedule to modify that code to be suitable for the new system. When code was designed to reuse, it was much easier to insert into a new implementation.

Documentation Several experts mentioned the need to have good documentation on the products they plan to use. This documentation must include information about what the software does, so the developer knows what it is. This avoids having to dive into the code to figure it out. The documentation must capture enough metadata about the reuse artifact that a developer can determine in a reasonable amount of time whether that component is one worth investing in. The documentation must include changes made after design review and after tests, so that is an as built document rather than a build to document.

Experience Much code is so obscure, it requires individuals who have used that code to understand what it does, and if it needs to be changed, where it needs modification.

Similar Product or Environment Several experts mentioned how success is determined by the similarity of the new deployed environment is. Reuse products may behave differently in a new environment.

Consistent baseline A consistent baseline helps control the development process and avoids churn that can occur when the baseline is constantly changing.

Automated testing The automated tests can be conducted without the developer. Not only does this free the developer to do more development, it allows the tests to be rerun in a consistent manner so results are comparable.

Graphical Representation The graphical representation allows the developer to quickly understand the capabilities of the code, the context, and visualize the differences for his own system. A good model can add or remove specific elements without impacting the rest of the model.

Product Line Much of the system can be reused. The product line has procedures to add or remove to satisfy unique customer requirements.

Service Oriented Architecture A service-oriented architecture can modularize and define the interfaces, and support a nice interface between software applications and scientific algorithms.

6.4.5 Summary of Obstacles

From the semistructured interviews, we were also able to identify a number of technical obstacles to successful reuse:

Understanding Experts found that they often did not understand the reuse products they were inheriting or how much they would have to be modified to work in the new system. They often did not understand the limitations the new software would place on their solution. They requently did not understand the interfaces,the coding standards or the naming conventions. Often reuse products that appeared to be a good fit to their new system were, in fact, not well suited.

Complexity Often the imported reuse products were extremely complex. It was difficult to understand what was happening in the code and to trace the impacts of modification, especially in closely coupled code. It was difficult to get a mental picture of how the code worked.

Lack of Documentation The flip side of the success factor of good documentation was the obstacle of poor or no documentation. Many imported products came with little or no documentation, so the developers could not figure out what the code was doing, what the required interfaces were to other applications or the hardware, or how the threads traced through the software.

Forking When a new system takes over a reuse product and modifies it into a variant of that baseline, it duplicates the total life cycle cost of maintaining what becomes unique software. Then there are two copies of the code. When developers run into problems in the code, since they've forked, they miss the benefit of getting the fixes for the problems that were in the code, they miss advances when the design is updated. Thus, benefits of a common set of software are lost.

Platform Dependence If software runs on a certain server and operating system, the developer is confined to using that server and operating system, limiting solution choices. The machine architecture influences what is and isn't reusable. The new platform may not have the same computer language and differences with compilers, development environments, means the code has to change. Typically different hardware has different control and status interfaces and messages and response requirements. For scientific algorithms, when they are moved from one type of the processor to another, all the underlying mathematical libraries changed.

Missed Opportunities Often, developers don't even try to reuse. they often discover missed opportunities for reuse, not knowing that somebody on another program had built

something similar they could have reused. They need access to the information to know what there is to reuse and where it is, what requirements it matches, and the design.

Performance Goals Timelines are based on the requirements, and how the component supports that performance. When the reuse doesn't meet performance requirements, developers have to evaluate what has to be done to that reused software so that it will meet performance requirements, and if they have to redesign it, it may not be worth the reuse.

Difficulty Developers often think thatreuse is going to be more effort than to just start out from scratch, that they are wasting time and it's more frustration than it's worth. It never goes quite as smoothly as expected. It can be something as simple as the way they name the variables. The code can be more trouble than it's worth.

Insert Defects Sometimes the process of modifying can insert defects.

Certification Process Sometimes code is not certified, or is certified to an obsolete or different standard. the code has to be rewritten to meet the requirements in the new standard.

Maintenance Maintaining old but modified code can be difficult. If the code is in an older language, it may be hard to find people who are familiar with that code. It may be difficult to identify insertion points, especially if the code is not well documented.

6.4.6 Developing a New Approach

We noticed that the biggest differences between reuse embedded and nonembedded systems lay in the reuse of code. Embedded systems developers in particular found that modifying code or living with its limitations caused the most trouble. We wondered whether, if this problem was removed, we could develop an approach that would work for both embedded and nonembedded systems to maximize the benefits of reuse.

To realize the full benefit of reuse and minimize the obstacles, we suggest a different approach, an ontology. This ontology considers the benefits of each development approach and the artifacts and minimizes the detriments. It also applies the technical success factors while minimizing the technical obstacles. The structure of the ontology and the process are contained in Figures 6.1 and 6.2, respectively.

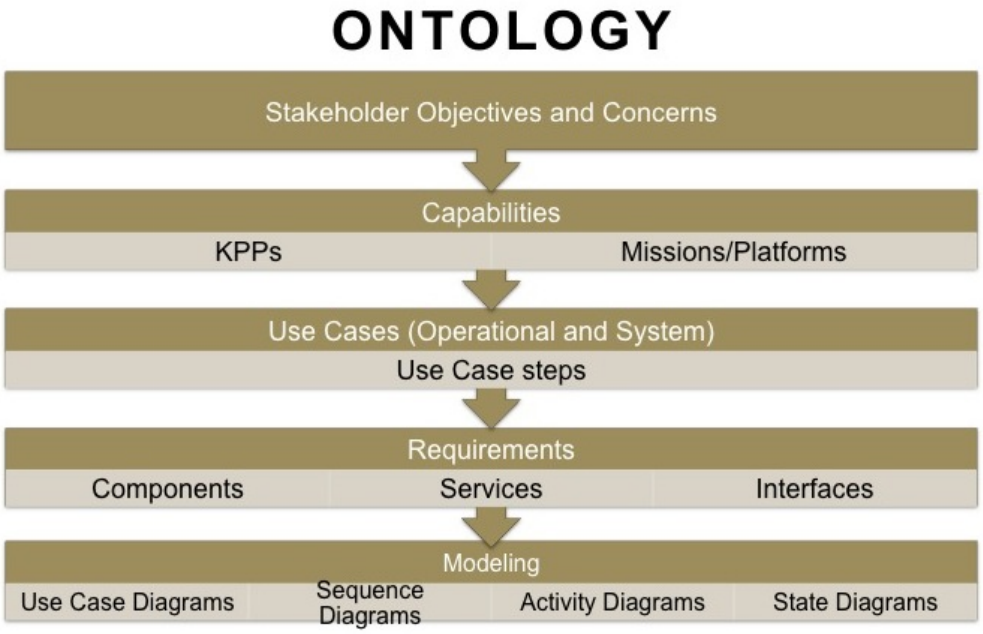


Figure 6.1: Structure of the Ontology.

The first obstacle that needs to be addressed is the problem of the solution not answering the objectives or providing a suboptimal solution, i.e. fit. To answer this obstacle, we develop a dataset of operational objectives for which solutions are already developed. These objectives include the operating environment, behaviors and nonfunctional requirements, such as security, safety and performance. If a developer has an objective to meet, the developer will have to develop a solution to that objective, however, many objectives will already have an answer. Those objectives will be linked to a database of operational capabilities that meet those objectives. Operational capabilities are linked to operational use cases to help identify operational requirements. These use cases lead to to a database of operational requirements that satisfy those capabilities as described in the use cases. The

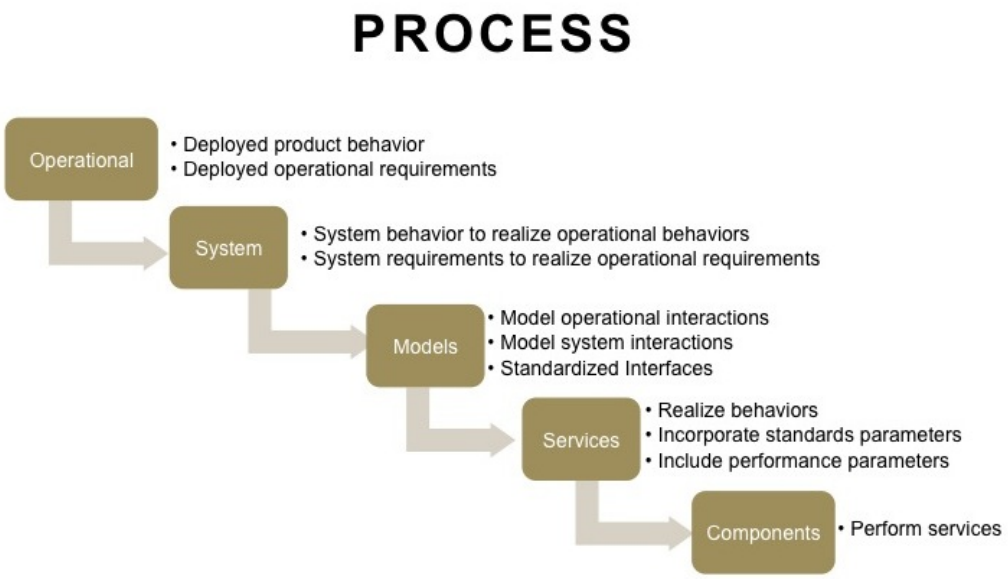


Figure 6.2: Process for Creating and Using the Ontology

operational requirements are linked to operational models that include operational rules, operational state models and operational activity diagrams or operational sequence diagrams. Thus we have the operational views and capabilities views called out in architecture frameworks.

The next step is to identify systems objectives to satisfy the operational objectives. These systems objectives are linked to the operational objectives mentioned above. The developer will be able to select among various systems solutions to satisfy his operational objectives originally selected. Again, these systems objectives include the system environment, systems behaviors and nonfunctional requirements. These systems objectives are then, as with the operational objectives, linked to systems capabilities, systems use cases, systems requirements and systems models. Thus, we have the systems views identified in architecture frameworks. The ontology can realize the benefits offered by the success factor of using a product line by storing all potential options for the product line that could be selected by a customer and generating most, if not all, of the system through selection of the customer preferences. By applying this approach, we remove many of the obstacles, such as "fit," understanding, and complexity (the complexity is hidden from the developer).

At this point, we create three databases, one containing components, one containing a catalog of services that satisfy specific system requirements and one containing interfaces to allow integration. The center of these is the services database, which identifies services to satisfy system requirements. The services are linked to any components that provide a given service as well as to any interfaces the service uses to integrate to a system. This imports the success factor of the service oriented architecture. The components database includes both COT/GOTS products and existing code modules that perform various functions. For COTS products, the database contains the name of the provider, any documentation about functional specifics, nonfunctional requirements satisfied, certifications, interfaces available and information about anticipated upgrades. Code products include appropriate documentation as well as the platforms it can run on. They can also include test drivers, automated tests, data files and any other associated test products. Interfaces include the interface specifications, whether proprietary or open, the year of issue, and if they have been superceded. As COTS and components are developed or modified, they are added to the database to provide the developers with the ability to select new products or the older products, if they prefer. Through this, we access the success factors of control, good documentation, a consistent baseline, and automated testing. This mitigates the obstacle of poor fit, poor documentation, inserting defects, missed opportunities, and performance goals.

Finally, we create a database of models that satisfy requirements. These models contain any diagrams, the services from the services database, the components from the components database and the interfaces from the interface database. These models can be modified as needed to tailor the new development to actual requirements, or new ones can be created if no similar model is available. The new and modified models are also stored in the database, to offer the developer more options for his/her solution. The models include platform specific information with the ability to select the target platform. They also include information about nonfunctional requirements and the ability to select/add nonfunctional requirements in the model. The models are able to generate new code if it is needed, as

well as generate documentation. In cases where existing code needs modification, it is recommended that instead it be modeled and autogenerated and automatically tested. This mitigates the obstacle of modification to fit the solution. This brings in the success factor of a graphical representation and good documentation. It mitigates the obstacle of complexity, understanding, being easier to build from scratch, obsolescence and forking. It also mitigates the problem of having to redevelop software products to satisfy nonfunctional requirements.

The developer enters the ontology through a wizard. This wizard allows the developer to select his operational objectives. The database uses the operational objectives to identify operational and systems capabilities and requirements, and return to the developer options in models, services and components, as well as identifying the objectives for which the tool has no solution. The wizard allows the developer to select artifacts from the options identified by the wizard to import into his/her new system, or to reject them and create new ones if the available artifacts are not suitable. The developer would turn to the models database to create new models or to modify existing models tailored to the system nuances. The models database could then autogenerate new code and enter it into the components and services database with accompanying documentation. If the new code needs optimizing, the developer can optimize in the components database. This way, the developer can concentrate his/her efforts on those objectives that have not already been solved. When new solutions are developed, they can be added to this central database and made available for use in other systems. The wizard can also be accessed by product providers who wish to add their products into the appropriate databases and make them available to the developers, as shown in Figure 6.3.

For future work, we propose a project that implements this ontology. We would develop metrics to determine the success of the ontology against the cost of its development and maintenance. From this, the hope is that the expectations of savings through reuse could be realized.

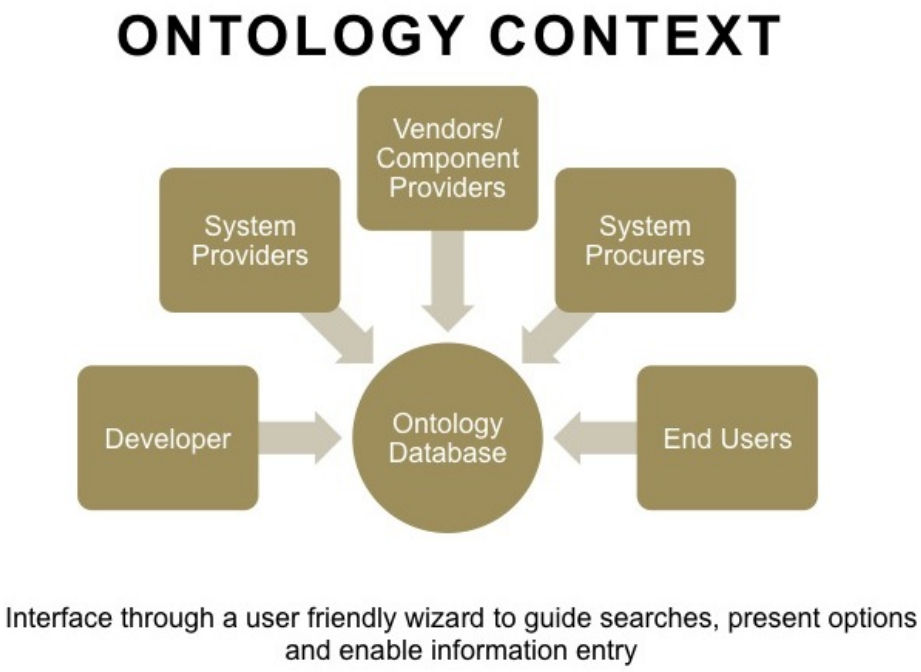


Figure 6.3: Context of the Ontology

6.5 Threats to Validity

Because this is a mixed methods study, we need to look at threats to validity both from a quantitative and qualitative perspective.

6.5.1 Quantitative Threats to Validity

Wohlin et. al. [129] name four quantitative threats to validity: conclusion validity, internal validity, construct validity and external validity.

Conclusion Validity is concerned with the relationship between the treatment and the outcome. One of our threats to conclusion validity is the low statistical power of these tests. This is the result of a fairly low sampling and the large number of variables. We recommend repeating this study with more observations and the removal of the variables that indicated little impact on the treatments.

Internal Validity is concerned with whether there is in fact a causal relationship and not influenced by a factor that has not been measured. A threat to conclusion validity in this study is the self selection in the survey and, to some degree, in the semistructured interviews. Subjects may have chosen to take the survey because they had an agenda related to reuse. This has been mitigated by the observation that the answers indicate a nearly equal number of those who have issues with reuse and those who are enthusiastic about reuse. Experts may have agreed to be interviewed because they have some frustrations with reuse they are anxious to share.

Construct Validity is concerned with whether the treatment is in fact related to the cause, and that the outcome does in fact reflect the construct. One construct threat to validity in this experiment is the interaction of the different variables. For example, while we were intending to study the differences between embedded and nonembedded systems, the dependent variables of development approach may have had an impact on the artifacts selected. We mitigate this threat by studying the variable types separately.

External Validity is concerned with the ability to extend conclusions outside of the experiment. One major consideration is that many of the projects reported on have been ongoing for many years. Another is that the projects are either very large systems themselves, or research and development to be inserted into very large systems. Whether smaller projects would experience different results is not clear.

6.5.2 Qualitative threats to validity

Since some of the information collected is qualitative, we assess our approach with respect to: descriptive validity, interpretive validity, theoretical validity, generalizability, and evaluative validity [90].

Descriptive Validity relates to the quality of what the subject reports having seen, heard or observed. Here, not only are these observations subjective, since the projects are

so large the respondents may see only a small part of the effort. Their observations may not reflect the whole project.

Interpretive Validity is concerned with what objects, events, and observations mean to the subjects. In our case, impartiality of reporting is uncertain.

Theoretical Validity refers to an account's validity as a theory of some phenomenon. It depends on the validity of the construct of the experiment and on the validity of the interpretation or explanation of the observations. It also depends on whether there is consensus within the community about terms used to describe context, events and outcomes. With the number of similar but not identical metrics presented, and ways of measuring success, there is a threat to validity concerning the similarity or difference of the perceived value of reuse. This was mitigated by using a common scoring system, resulting in ordinal rather than ratio metrics.

Generalizability "refers to the extent to which one can extend the account of a particular situation or population to other persons, times, or settings than those directly studied [90]." Here, the threat to validity consists of the fact that the research was performed within the confines of one corporation. While the corporation consists of several companies and cultures, there may also be an influence of the corporation itself. There also could be proprietary information that was not discussed. In addition, there may be a tendency to report successes and to keep working on or terminate efforts that have failed, success may be overreported. On the other hand, there might also be a tendency to report failures and the failures may be overreported. There may be an underreporting of results that were neither great successes nor failures. However, the results we obtained do not seem to indicate either situation.

Evaluative Validity refers to the evaluative framework in deciding whether the reuse was, in fact, successful or not, and if so how much. The frameworks were likely to have

differed in the different projects because the contexts were different. This was mitigated by considering the subjects' report of both reuse and the subjects' descriptive observations upon which the reports are based.

One final threat to validity in this research is the ability to replicate the studies. In the case of both the survey and the semistructured interviews, most researchers would have difficulty gaining access to the subjects and obtaining frank responses.

6.6 Conclusions

Today, embedded systems are slowly reengineering their existing systems and developing new systems with frameworks and models. This is an important development, since the architecture and design phases of the lifecycle are much longer than the implementation phase. Thus the architecture and design phases offer more opportunities for savings. Many of the architecture models have the capability to automatically generate code (autogen or autocode).

In the series of semistructured interviews, many interviewees indicated that while earlier, when reuse was mentioned, they had only considered code, today they are reusing many different artifacts. These artifacts include requirements, architecture, design and design products, models, documentation, and test products. When asked to rate these artifacts in terms of reuse effectiveness (defined as a series of desired outcomes), some put test products first, others put architecture first. However, all but two placed code last. One progressive embedded software subject matter expert (SME) indicated that over time, he expected most code to be generated automatically, and the only code that would be hand written would be code that needs to be optimized to the platform.

The move to standardize platforms is having a major positive impact on software reuse. Design models are able to include the platform parameters in the model itself, and then generate the code appropriate to the platform. Tweaking the code generator in the model allows the model to optimize the generated code to the platform, with less need for modi-

fication. One expert predicts that in time, the only coders will be those who specialize in optimizing very specialized code.

As more and more assets are designed to be reusable, companies are investing in reference architectures and searchable libraries. We envision these to become ontologies, which attach objectives to capabilities, the capabilities to requirements, the requirements to models, designs, test cases, and so on. These ontologies include the documentation A developer could select the objectives of his project, and with it select from various reusable solutions. The selected solution would be available to the developer, and the developer would only have to fill in the holes. So the future of software reuse in embedded systems will include:

- An increase in the reuse of architecture models and design models with code generation capabilities.
- An increase in the reuse of test products.
- A significant decrease in the reuse, or even the development, of code.
- An increased emphasis on parameter driven models to select from standardized platforms.
- An increased emphasis on full documentation from build to specifications to as built documents.
- Design for reuse as the norm rather than the exception.
- Infrastructures for supporting reuse integrating tools, techniques, methods, policies, and incentives.
- A new way of thinking, with emphasis on objectives and outcomes rather than on development.

Reuse approaches for over 20 years have promised lower costs for higher quality products with shorter time to deliver [4]. Customers have increasingly required reuse, management

hoped to reap its promised benefits, researchers tried to demonstrate them. The Information Technology Management Reform Act of 1996, also known as the Clinger-Cohen act [22], practically mandates reuse through the Performance- And Results-Based Management initiative. However, benefits of reuse are in no way assured, even with the best of intentions. This paper tried to chronicle some of the experiences with reuse. First, reuse has to be paired with good engineering practices. Second, technical factors can be instrumental in success vs. failure. Third, in the defense and aerospace industry, bespoke systems may not be able to recoup reuse investment cost [85]. Fourth, a reuse mandate does not guarantee its successful execution.

As older projects either age out or are refactored, more should be architecture and design focused. This will facilitate reuse, not as much in code, but in the other artifacts that are more expensive to develop.

Some studies [93] have tried to summarize reuse experiences in industry, but with limited success due to the lack of quantitative data. We recommend pairing reuse with detailed measurement so that evaluating reuse success is no longer a matter of opinion.

We conclude that in today's environment, reuse in embedded systems does differ from reuse in nonembedded systems. However, we do not believe this will continue to be the case. As older systems are phased out and upgraded, and as platforms become standardized, embedded systems can move to more model based development and test. The importance of code reuse will decline, and eventually become a specialty rather than a fundamental part of reuse.

Reuse has an impact not only on the government, but also on universities and students, contractors, and vendors. The changing focus of reuse will change the focus of a developer's activites and needed skills. While it will always be important for universities to teach and students to learn coding and coding techniques, students will need to master modeling and architecture, of both systems and testing. Contractors will need to focus on hiring employees who are versed in modeling, as well as develop a set of model based metrics to aid in estimating. Vendors will need to develop products that interface with the standardized

platforms. There is an added benefit for vendors, because the standardized platforms open the market to new products.

In answer to the findings, we proposed the development of a framework of framework, an ontology. This ontology would contain, in a searchable library, objectives, use cases, requirements, architecture patterns, design patterns, models, interfaces, and test cases. It would also include components: encapsulated algorithms, platforms, and hardware with software (Off the Shelf(OTS)), source code (to be used unmodified) to make reusable products available, easy to find and easy to use. These elements would all be linked, so that when a developer searches on an objective, the appropriate requirements, architecture, design, models, interfaces and components would be recommended. Code not included in OTS or encapsulated algorithms would be generated by the models. By creating such an ontology, developers could take advantage of the benefits of each development approach and select the artifacts most suited to his project.

Bibliography

[1] S. Adolph. Whatever Happened to Reuse? *Dr. Dobbs.com*, November 1 1999. http://www.drdobbs.com/architect/184415752.

[2] R. Alur, T Dang, J. Esposito, Y Hur, F. Ivancic, V. Kumar, P. Mishra, G.J. Pappas, and O. Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proceedings of the IEEE*, 91(1):11 – 28, Jan 2003.

[3] American Psychological Association, Washington, DC. *Publication Manual of the American Psychological Association*, sixth edition, 2010.

[4] C Anderson and M Dorfman. *Aerospace Software Engineering: A Collection of Concepts*. Progress in Astronautics and Aeronautics. American Institute of Aeronautics and Astronautics, 1991.

[5] A Andrews, A Stefik, and J Varnell-Sarjeant. Comparing Reuse Strategies: An Empirical Evaluation of Developer Views. In *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th Annual*, July 2014.

[6] A Andrews and J Varnell-Sarjeant. Comparing Development Approaches and Reuse Strategies: An Empirical Evaluation of Developer Views from the Aerospace Industry. submitted.

[7] A Andrews and J Varnell-Sarjeant. Comparing Reuse in Different Development Approaches For Embedded vs Non-Embedded Systems. Accepted by Advances in Computing.

[8] A Andrews and J Varnell-Sarjeant. Reflections on the History of Software Reuse for Embedded Systems in the Defense and Aerospace Industry. submitted, 2013.

[9] R. Anguswamy and W. B. Frakes. A Study of Reusability, Complexity, and Reuse design Principles. In *Proceedings of the ACM-IEEE international symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 161–164, New York, NY, USA, 2012. ACM.

[10] B. Barlin and J. M. Lawler. Effective Software Reuse in an Embedded Real-time System. In *TRI-Ada '92: Proceedings of the Conference on TRI-Ada '92*, pages 281–287. ACM, 1992.

[11] V. R. Basili, M. V. Zelkowitz, D. I. Sjoberg, and A. J. Johnson, P.and Cowling. Protocols in the Use of Empirical Software Engineering Artifacts. *Empirical Software Engineering*, 12(1):107–119, 2007.

[12] S. Bhatia, C. Consel, and C. Pu. Remote Specialization for Efficient Embedded Operating Systems. *ACM Transactions Programming Language Systems*, 30(4):1–32, 2008.

[13] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston. *Object-Oriented Analysis and Design with Applications*. Pearson Education, Inc, third edition, 2007.

[14] L Brown. DoD Software Reuse Initiative: Vision and Strategy. Technical report, Department of Defense, 1225 Jefferson Davis Highway, Suite 910, Arlington, VA 22202-4301, July 1992.

[15] G. Brusa, Ma. L. Caliusco, and O. Chiotti. A Process for Building a Domain Ontology: An Experience in Developing a Government Budgetary Ontology. In *AOW '06: Proceedings of the Second Australasian Workshop on Advances in Ontologies*, pages 7–15. Australian Computer Society, Inc., 2006.

[16] N.B. Bui, L. Zhu, I. Gorton, and Y. Liu. Benchmark Generation Using Domain Specific Modeling. In *18th Australian Software Engineering Conference, 2007. ASWEC 2007.*, pages 169 –180, 10-13 2007.

[17] C. Bunse, H. Gross, and C. Peper. Embedded System Construction — Evaluation of Model-Driven and Component-Based Development Approaches. In MichelR.V. Chaudron, editor, *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 66–77. Springer Berlin Heidelberg, 2009.

[18] C. Wohlin and A. Amschler Andrews. Evaluation of three methods to predict project success: A case study. In *Proceedings of International Conference on Product Focused Software Process Improvement (PROFES05) LNCS-series*, pages 385–398, Oulu,Finland, 2005.

[19] X. Cai, M.R. Lyu, K. Wong, and R. Ko. Component-based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes. In *Proceedings of the Seventh Asia-Pacific APSEC 2000.*, pages 372 –379, 2000.

[20] N. Choi, I. Song, and H. Han. A Survey on Ontology Mapping. *SIGMOD Rec.*, 35(3):34–41, 2006.

[21] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[22] W. Clinger and W. Cohen. Information technology management reform act of 1996.

[23] CS Condon, S Seaman, S Kraft, V Basili, and Y Kim. Evolving the Reuse Process at the Flight Dynamics Division (FDD) of Goddard Space Flight Center. Technical report, NASA, 1997.

[24] R. Conn, S. Traub, and S. Chung. Avionics Modernization and the C-130J Software Factory. *Crosstalk:The Journal of Defense Software Engineering*, September 2001. http://www.crosstalkonline.org/storage/issue-archives/2001/200109/200109-0-Issue.pdf.

[25] J. W. Costello, A. B.and Osborne. Best practices in exploratory factor analysis: Four recommendations for getting the most from your analysis. *Practical Assessment, Research and Evaluation*, 10(7):1–9, 2005.

[26] E.S. de Almeida, A. Alvaro, D. Lucredio, V.C. Garcia, and S.R. de Lemos Meira. A Survey on Software Reuse Processes. In *IEEE International Conference on Information Reuse and Integration, Conference, 2005.*, pages 66 – 71, Aug. 2005.

[27] A.M. de Cima, C.M.L. Werner, and A.A.C. Cerqueira. The Design of Object-oriented Software With Domain Architecture Reuse. In *Proceedings of theThird International Conference on Software Reuse: Advances in Software Reusability, 1994*, pages 178 –187, 1-4 1994.

[28] Department of Defense. C4ISR Architecture Framework. Technical report, Department of Defense, 1996.

[29] Department of Defense. Department of Defense Architecture Framework 2.0. Technical report, Department of Defense, 2011.

[30] P. Devanbu, S. Karstu, W. Melo, and W. Thomas. Analytical and Empirical Evaluation of Software Reuse Metrics. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 189–199. IEEE Computer Society, 1996.

[31] E. W. Dijkstra. Letters to the Editor: Go To Statement Considered Harmful. *Communications of the ACM*, 1968.

[32] D. Dikel, D. Kane, and B. Loftus. A Case Study of Software Architecture Organizational Success Factors. In *Reuse'96*. Applied Expertise, July 1996.

[33] DoD Joint Technical Architecture. Department of Defense Joint Technical Architecture. Technical report, Department of Defense, 1996.

[34] DoD Joint Technical Architecture. Department of Defense Joint Technical Architecture. Technical report, Department of Defense, 2003.

[35] W.A Dos Santos and A. M da Cunha. An MDA Approach for a Multi-Layered Satellite On-Board Software Architecture. In *5th Working IEEE/IFIP Conference on Software Architecture, 2005.*, pages 253 –256, 2005.

[36] D. Eichmann. Factors in Reuse and Reengineering of Legacy Software. Technical report, Repository Based Software Engineering Program Research Institute for Computing and Information Systems, University of Houston, 1997.

[37] J. Estefan. Survey of Candidate Model-Based Engineering (MBSE) Methodologies. In *INCOSE Survey of MBSE Methodologies*, pages 1–70. International Council on Systems Engineering (INCOSE), 2008.

[38] M. Ezran, M. Morisio, and C Tully. Failure and Success Factors in Reuse Programs: A Synthesis of Industrial Experiences. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 681–682. ACM, 1999.

[39] N Ferreira, R J. Machado, and D Gasevic. An Ontology-Based Approach to Model-Driven Software Product Lines. In *ICSEA '09: Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, pages 559–564. IEEE Computer Society, 2009.

[40] W. Frakes and S. Isoda. Success Factors in Systematic Reuse. *IEEE Software*, 1994.

[41] W. B. Frakes and C. J. Fox. Modeling Reuse Across the Software Life Cycle. *J. of Systems and Software*, 30(3):295–301, 1995.

[42] W. B. Frakes and C. J. Fox. Sixteen Questions about Software Reuse. *Commun. ACM*, 38(6):75–ff., June 1995.

[43] W. B. Frakes and S. Isoda. Success Factors of Systematic Reuse. *IEEE Software.*, 11(5):14–19, 1994.

[44] W. B. Frakes and G. Succi. An Industrial Study of Reuse, Quality, and Productivity. *Journal of Systems and Software*, 57(2):99–106, 2001.

[45] W.B. Frakes. A Case Study of a Reusable Component Collection. In *Proceedings of 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology, 2000*, pages 79–84, 2000.

[46] JJ Francis, M Johnston, C Robertson, L Glidewell, V Entwistle, MP Eccles, and JM Grimshaw. What is an Adequate Sample Size? Operationalising Data Saturation for Theory-Based Interview Studies. *Psychology and Health*, 25:1229–1245, December 2010.

[47] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12(6):17 –26, Nov 1995.

[48] D. Garlan, R. Allen, and J.M. Ockerbloom. Architectural Mismatch: Why Reuse is Still So Hard. *IEEE Software*, 26(4):66–69, 2009.

[49] J.M. Garrido. *Object Oriented Simulation: A Modeling and Programming Perspective*. Springer, 2009.

[50] R. Gerard, R.R. Downs, J.J. Marshall, and R.E. Wolfe. The Software Reuse Working Group: A Case Study in Fostering Reuse. In *IEEE International Conference on Information Reuse and Integration, 2007*, pages 24 –29, 13-15 2007.

[51] B G Glaser and A L Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Observations (Chicago, Ill.). Aldine, 1967.

[52] P. A. Glasgow. Fundamentals of Survey Research Methodology. Technical report, Mitre Corporaton, Washington C3 Center, McLean, Virginia, April 2005.

[53] G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P. G. Paulin. *Embedded Software in Real-time Signal Processing Systems: Design Technologies*, pages 433–451. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[54] B. Graaf and H. Van Dijk. Evaluating an Embedded Software Reference Architecture - Industrial Experience Report. In *In Proc. 9th European Conf. Software Maintenance and Reengineering (CSMR 2005). IEEE CS*, pages 354–363, 2005.

[55] J. Guojie, Y Baolin, and Z. Qiyang. Enhancing Software Reuse through Application-level Component Approach. *Journal of Software*, 6(3):374 – 385, 2011.

[56] A. Gupta. *The Profile of Software Changes in Reused vs. Non-Reused Industrial Software Systems.* PhD thesis, Norwegian University of Science and Technology, 2009.

[57] A Gupta, J Li, R Conradi, H Ronneberg, and E Landre. A Case Study Comparing Defect Profiles of a Reused Framework and of Applications Reusing It. *Empirical Software Engineering*, 14(2):227–255, 2009.

[58] W Ha, H Sun, and M Xie. Reuse of Embedded Software in Small and Medium Enterprises. In *2012 IEEE International Conference on Management of Innovation and Technology (ICMIT)* , pages 394–399, 2012.

[59] P. A. V. Hall. Architecture-driven Component Reuse. *Information and Software Technology*, 41(14):963 – 968, 1999.

[60] S. Hallsteinsen and M. Paci. *Experiences in Software Evolution and Reuse: Twelve Real World Projects*, volume 1. Springer, 1997.

[61] M.C. Harrell, M.A. Bradley, Rand Corporation, and National Defense Research Institute (U.S.). *Data Collection Methods: Semi-Structured Interviews and Focus Groups: Training Manual.* Technical report (Rand Corporation). RAND Corporation, 2009.

[62] E Heinz, P Lukowicz, W. F. Tichy, and L Prechelt. Experimental Evaluation in Computer Science: A Quantitative Study. In *Journal of Systems and Software*, volume 28-1, pages 9–18, Jan 1995.

[63] S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions Software Engineering Methodology*, 6(2):111–140, 1997.

[64] R. Holmes and R. J. Walker. Systematizing Pragmatic Software Reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4):20:1–20:44, February 2013.

[65] IEEE Computer Society. IEEE 1517 - 2010 Standard for Information Technology - System and Software Lifecycle Processes - Reuse Processes. Technical report, Software and Systems Engineering Standards Committee, 2010.

[66] N. Ilk, J. Zhao, P. Goes, and P. Hofmann. Semantic Enrichment Process: An Approach to Software Component Reuse in Modernizing Enterprise Systems. *Information Systems Frontiers*, 13:359–370, 2011.

[67] International Organization for Standardization. Information Technology - Programming Languages, Their Environments and System Software Interfaces - Language-Independent Datatypes 1996. Technical report, ISO/IEC TR 18015:1996(E), 1996.

[68] International Organization for Standardization. Information Technology - Programming Languages, Their Environments and System Software Interfaces - Language-Independent Datatypes Technical Report on C++ Performance 2006. Technical report, ISO/IEC TR 18015:2006(E), 2006.

[69] S. Isoda. Experiences of a Software Reuse Project. *Journal of Systems and Software*, 30(3):171–186, 1995.

[70] E. K. Jackson and J. Sztipanovits. Towards a Formal Foundation for Domain Specific Modeling Languages. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pages 53–62. ACM, 2006.

[71] I. Jacobson, M. Griss, and P. Jonsson. Making the Reuse Business Work. *Computer*, 30(10):36–42, 1997.

[72] J Jiao, T Simpson, and Z Siddique. Product Family Design and Platform-based Product Development: a State-of-the-art Review. *Journal of Intelligent Manufacturing*, 18:5–29, 2007.

[73] R. Kamalraj, Dr. Kannan A. R., and P. Ranjani. Stability-based Component Clustering for Designing Software Reuse Repository. *International Journal of Computer Applications*, 27(3):33–36, August 2011. Published by Foundation of Computer Science, New York, USA.

[74] E. A. Karlsson, editor. *Software reuse: a holistic approach.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[75] K Kim, H Kim, and W Kim. Building Software Product Line from the Legacy Systems: Experience in the Digital Audio and Video Domain. In *11th International Software Product Line Conference, 2007*, pages 171 –180, Oct 2007.

[76] Y. Kim and E. A. Stohr. Software Reuse: Survey and Research Directions. *Journal of Management Information Systems*, 14(4):113–147, 1998.

[77] B. A. Kitchenham. Procedures for Performing Systematic Reviews. Technical Report TR/SE-0401, Software Engineering Group, Department of Computer Science, Keele University, NSW 1430, 2004.

[78] V Koppen, N Siegmund, M Soffner, and G Saake. An Architecture for Interoperability of Embedded Systems and Virtual Reality. *IETE Technical Review*, 26:350–6, 2009.

[79] V. Krishnan, R. Singh, and D. Tirupati. A Model-Based Approach for Planning and Developing A Family of Technology-Based Products. *Manufacturing and Operations Management*, 1(2):132–156, Spring 1999.

[80] M. Kuhlemann, D. Batory, and S. Apel. Refactoring Feature Modules. In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, ICSR '09, pages 106–115, Berlin, Heidelberg, 2009. Springer-Verlag.

[81] C. F. J. Lange. Model Size Matters. In *Proceedings of the 2006 international conference on Models in software engineering*, MoDELS'06, pages 211–216, Berlin, Heidelberg, 2006. Springer-Verlag.

[82] E. A. Lee. Embedded Software. In *Advances in Computers*, pages 1–34. Academic Press, 2002.

[83] E. A. Lee. CPS Foundations. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 737–742. ACM, 2010.

[84] S. Lee, H. Ko, M. Han, D. Jo, and K. Jeong, J.and Kim. Reusable Software Requirements Development Process: Embedded Software Industry Experiences. In *ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference*, pages 147–158. IEEE Computer Society, 2007.

[85] N. G. Leveson and K. A. Weiss. Making Embedded Software Reuse Practical and Safe. *SIGSOFT Software Engineering Notes*, 29(6):171–178, 2004.

[86] J. Li, R. Conradi, P. Mohagheghi, O. A. Saehle, O. Wang, E. Naalsund, and O. A. Walseth. A Study of Developer Attitudes to Component Reuse in Three IT Companies. In *Product Focused Software Process Improvement*, volume 3009 of *Lecture Notes in Computer Science*, pages 538–552. Springer Berlin / Heidelberg, 2004.

[87] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Softw.*, 11(5):23–30, September 1994.

[88] W. C Lim. *Managing Software Reuse: a Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components*. Prentice Hall, 1998.

[89] D Lucredio, E Santana de Almeida, and R Fortes. An Investigation on the Impact of MDE on Software Reuse. *2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 101–110, 2012.

[90] J. A Maxwell. Understanding and Validity in Qualitative Research. *Harvard Educational Review*, 62(3):279–300, 1992.

[91] Dorothy McKinney. A quick history of reuse in xxx corporation. Internal Corporate Document.

[92] A. Mili, R. Mili, and R. T. Mittermeir. A Survey of Software Reuse Libraries. *Annals of Software Engineering*, 5:349–414, 1998.

[93] P. Mohagheghi and R. Conradi. Quality, Productivity and Economic Benefits of Software Reuse: a Review of Industrial Studies. *Empirical Software Engineering.*, 12(5):471–516, 2007.

[94] P. Mohagheghi and R. Conradi. An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product. *ACM Transactions Software Engineering Methodology*, 17(3):1–31, 2008.

[95] M. Morisio, M. Ezran, and C. Tully. Success and Failure Factors in Software Reuse. *IEEE Transactions on Software Engineering*, 28(4):340–357, 2002.

[96] NASA. NASA Software Engineering Requirements Appendix B. NASA Procedures and Guidelines, NASA, 2004.

[97] D. L. Nazareth and M. A. Rothenberger. Assessing the Cost-effectiveness of Software Reuse: A Model for Planned Reuse. *Journal of System Software*, 73(2):245–255, 2004.

[98] M. C. Ohlsson, A. Von Mayrhauser, B.McGuire, and C. Wohlin. Code decay analysis of legacy software through successive releases. In *Proceedings of the IEEE Aerospace Conference*, pages 69–81, 1999.

[99] M.C. Ohlsson and C. Wohlin. Identification of green, yellow, and red legacy components. In *Procs. International Conference on Software Maintenance*, pages 6–15, 1998.

[100] Open Systems Joint Task Force. A modular open systems approach (mosa) to weapon system acquisition executive summary. Technical report, Department of Defense, 2004.

[101] A. Orrego, T. Menzies, and O. El-Rawas. On the Relative Merits of Software Reuse. In *ICSP '09: Proceedings of the International Conference on Software Process*, pages 186–197. Springer-Verlag, 2009.

[102] L. Pareto, M. Staron, and P. Eriksson. Ontology Guided Evolution of Complex Embedded Systems Projects in the Direction of MDA. In *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 874–888. Springer Berlin / Heidelberg, 2010.

[103] P. G. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens. Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends. In *Proceedings of the IEEE*, volume 85, pages 419–435, 1997.

[104] Y. Peng, C. Peng, J. Huang, and K. Huang. An Ontology-Driven Paradigm for Component Representation and Retrieval. In *Ninth IEEE International Conference on Computer and Information Technology, 2009. CIT '09.*, volume 2, pages 187 –192, 11-14 2009.

[105] J. S. Poulin. The Business Case for Software Reuse: Reuse Metrics, Economic Models, Organizational Issues, and Case Studies. In Maurizio Morisio, editor, *Reuse of Off-*

the-Shelf Components, volume 4039 of *Lecture Notes in Computer Science*, pages 439–439. Springer Berlin Heidelberg, 2006.

[106] M. Raatikainen, T. Soininen, T . Mannisto, and A. Mattila. A Case Study of Two Configurable Software Product Families. In Frank van der Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 403–421. Springer Berlin / Heidelberg, 2004.

[107] D. C. Rine. Supporting Reuse with Object Technology. *Computer*, 30(10):43–45, 1997.

[108] D. C. Rine and R. M. Sonnemann. Investments in Reusable Software. A Study of Software Reuse Investment Success Factors. *Journal of System Software*, 41(1):17–32, 1998.

[109] D.C.; Nada N Rine. An Empirical Study of a Software Reuse Reference Model. *Information and Software Technology*, 42(1):47–65, Jan 2000.

[110] M. A Rothenberger. Systems Development with Systematic Software Reuse: An Empirical Analysis of Project Success Factors, 1999.

[111] M. A. Rothenberger, K. J. Dooley, U. R. Kulkarni, and N. Nada. Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices. *IEEE Transactions in Software Eng.*, 29(9):825–837, 2003.

[112] P Runeson, M Höst, A Rainer, and B Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley and Sons, 2012.

[113] Thomas L. Saaty. Priority setting in complex problems. *Engineering Management, IEEE Transactions on*, EM-30(3):140–155, 1983.

[114] T. Schetter, M. Campbell, and D. Surka. Comparison of Multiple Agent-Based Organisations for Satellite Constellations (TechSat21). *Artificial Intelligence*, 145(1-2):147 – 180, 2003.

[115] M. Schwartz. The Nunn-McCurdy act: Background, Analysis, and Issues for Congress. Technical report, Congressional Research Service, 2010.

[116] SEI. Software Product Lines. Technical report, Software Engineering Institute (SEI), http://www.sei.cmu.edu/productlines/, 2010.

[117] S. G. Shiva and L. A. Shala. Software Reuse: Research and Practice. In *Fourth International Conference on Information Technology, 2007. ITNG '07*, pages 603 –609. IEEE, Apr. 2007.

[118] J Sprinkleand, J. M Eklund, H Gonzalez, E I Groetli, B Upcroft, A Makarenko, W Uther, M Moser, R Fitch, H Durrant-Whyte, and S. S Sastry. Model-Based Design: a Report from the Trenches of the DARPA Urban Challenge. *Software and Systems Modeling*, 8:551–566, 2009.

[119] R Studer, V R Benjamins, and D Fensel. Knowledge Engineering: Principles and Methods. *IEEE Transactions on Data and Knowledge Engineering*, 25, 1998.

[120] T Sullivan, D Sather, and R Nishinaga. A Flexible Satellite Command and Control Framework, Oct 27, 2009.

[121] D.M. Surka, M.C. Brito, and C.G. Harvey. The Real-time ObjectAgent Software Architecture for Distributed Satellite Systems. In *IEEE Proceedings of the Aerospace Conference, 2001*, volume 6, pages 2731 –2741, 2001.

[122] T. M. Khoshgoftaar and D. L. Lanning . Are the principal components of software complexity stable across software products? In *Procs. International Symposium on Software Metrics*, pages 61–72, October 1994.

[123] T. M. Khoshgoftaar and E.B. Allen and N. Goel and A. Nandi and J. McMullan . Detection of software modules with high code debug churn in a very large legacy system. In *Procs. International Symposium on Software Reliability Engineering*, pages 364–371, October 1996.

[124] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[125] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of Partial Behavior Models from Properties and Scenarios. *IEEE Transactions on Software Engineering*, 35(3):384 –406, May-June 2009.

[126] C.A. Welty and D.A. Ferrucci. A Formal Ontology for Re-use of Software Architecture Documents. In *14th IEEE International Conference on Automated Software Engineering, 1999*, pages 259 –262, Oct 1999.

[127] S. Winkler and J. Pilgrim. A Survey of Traceability in Requirements Engineering and Model-Driven Development. *Software System Modeling*, 9(4):529–565, 2010.

[128] M. Winter, T. Genler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, P. Moeller, C. Stich, and B. Schoenhage. Components for Embedded Software - The PECOS Approach. In *CASES '02 Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 19–25. ACM Press, 2002.

[129] C Wohlin, P Runeson, M Höst, M. C. Ohlsson, B Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.

[130] H Yan, W Zhang, H Zhao, and H Mei. An Optimization Strategy to Feature Models,Äô Verification by Eliminating Verification-Irrelevant Features and Constraints. In S H. Edwards and G Kulczycki, editors, *Formal Foundations of Reuse and Domain Engineering*, volume 5791 of *Lecture Notes in Computer Science*, pages 65–75. Springer Berlin Heidelberg, 2009.

[131] R. K Yin. *Case Study Research Design and Methods*. SAGE Publications, 4th edition, October 2008.

[132] T. Young. Report of the Defense Science Board/ Air Force Scientific Advisory Board Joint Task Force on Acquisition of National Security Space Programs. Technical report, Office of the Under Secretary of Defense For Acquisition, Technology, and Logistics, Washington, D.C. 20301-3140, May 2003.

[133] C. Zannier, G. Melnik, and F. Maurer. On the Success of Empirical Studies in the International Conference on Software Engineering. In *Proceedings of the 28th International Conference on Software Engineering*, pages 341–350, New York, NY, USA, 2006. ACM.

[134] D.H. Zhang, Jing B. Z., M Luo, Y Tang, and L. Q Zhuang. A Reference Architecture and Functional Model for Monitoring and Diagnosis of Large Automated Systems. In *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation, 2003.*, volume 2, pages 516 – 523 vol.2, 16-19 2003.

[135] G. L. Zuniga. Ontology: Its Transformation from Philosophy to Information Systems. In *FOIS '01: Proceedings of the International Conference on Formal Ontology in Information Systems*, pages 187–197, New York, NY, USA, 2001. ACM.

Appendix A

Years of Publication

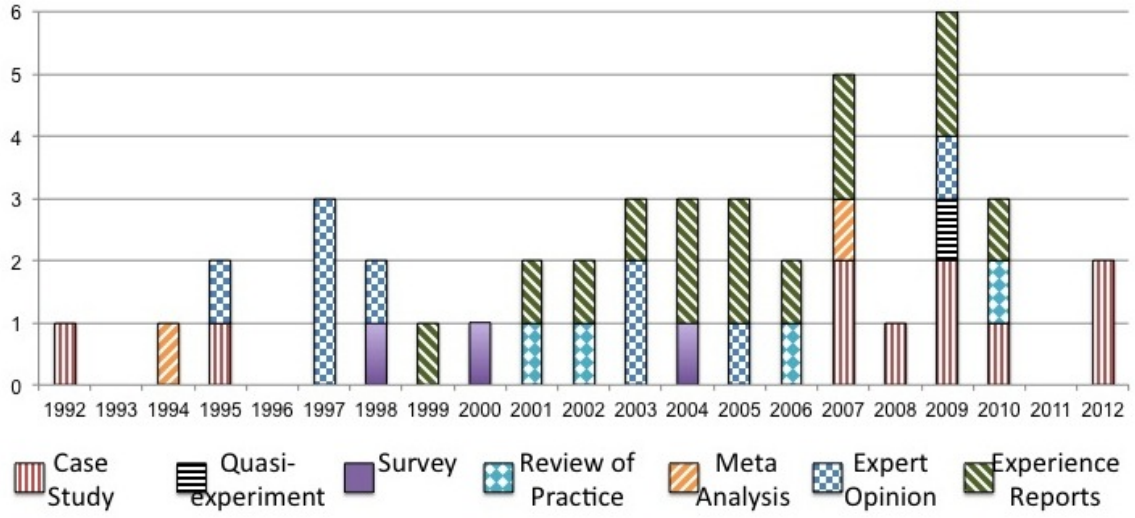


Figure A.1: Type of empirical study by year

One question was how empirical results on development strategies was evolving over time. Our study collected research since 1992. Empirical research on reuse increased since 2001 after a short spike in the late 1990s. Nineteen of the 24 case studies were conducted since 2002, as well as three of the five surveys. While in 1997 all of the empirical papers on development strategies were expert opinion, in 2007 and 2009 (the years with the most empirical papers on reuse) the types of studies were fairly well divided among case studies, experience reports and expert opinion with one each of review of practice, metaanalysis, and quasi-experiment. Even in 2010 and 2012, the studies were case studies and review of

practice, with only one experience report and no expert opinions. The empirical studies have become more focused on data and less on opinion. Figure A.1 illustrates this trend.

Another question was whether the development approaches using reuse were changing, or at least whether studies of development strategies were evolving. Empirical studies on product line reuse has been steady at one or two papers a year since 2003. However, the number of papers analyzing component based and model based reuse increased to a total of nine in 2009. Research on product lines has been fairly steady since 2002. There have also been a number of studies that investigated multiple development approaches. Ontologies have begun to get attention lately as well. On the other hand, the number of empirical papers written on reuse whose approach is unspecified has decreased to only one in each of 2008, 2009 and 2010. This shows that empirical research on reuse is beginning to focus on specific development strategies. Figure A.2 shows empirical studies by reuse in the development approaches by year.

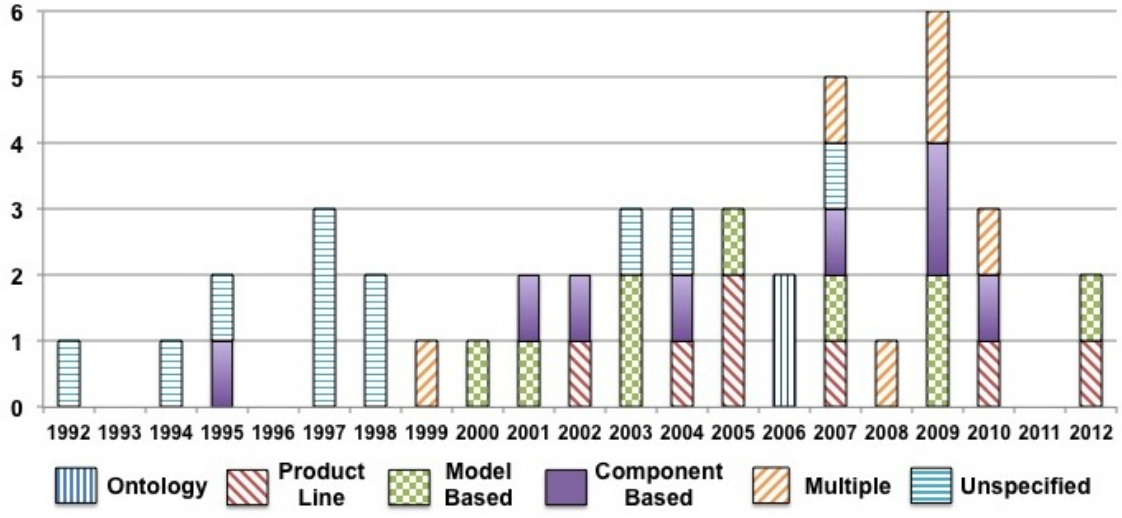


Figure A.2: Empirical studies of reuse by development approach by year

Appendix B

The Survey

The Survey The introduction:

This is an anonymous survey for all company systems and software engineers to determine various experiences we have had with different types of reuse. This information can be used to help analyze and apply best practices for reuse. It can also be used as data to include in proposals, for research papers such as doctoral dissertations and masters theses.

For purposes of this survey:

- A strategy is the choice of approach or combination of approaches the program makes to employ reuse
- An approach is the one of the development methods that allows for reuse (such as those four listed below):
  - Model-based reuse is reuse that is based on reusing models created on other programs or components.
  - Component-based is reuse based on already developed components or designed for reuse on a component basis.
  - Product line reuse is reuse based on a standardized but tailorable product line.
  - Ad Hoc is reuse that the engineer is familiar with, that happens to meet a requirement but was not designed for reuse.

If you have been involved in more than one project that employed reuse and had different experiences you would like to share, please respond once per program. In these cases please respond from the standpoint of that program and your experiences there. Summary results (and the resulting papers) will be posted here. We hope the results provide guidance in our future practices and help identify the best strategies and approaches for different types of programs We hope this survey can be used as a first step to assess what works and what does not work in reuse. All surveys used for academic research need to make the following disclaimer: Participation is voluntary. You can delete your response at any time. You can respond without identifying yourself.

Respondent Information

RQ-1 The purpose of these questions is to correlate reuse experience with the type of engineer (ie hardware, software, systems), the company (which corresponds to the types of programs and the culture), and the experience level of the engineer.

- 1. What type of engineer are you?
  - a. Systems
  - b. Software
  - c. Software Systems
  - d. Specify your own answer
- 2. What company and location do you work for?
- 3. How many years of experience do you have with system or software development?
  - a. 0-5 years
  - b. 6-10 years
  - c. More than 10 years
- 4. How many years of experience do you have with incorporating reuse into programs
  - a. 0-5 years
  - b. 6-10 years
  - c. More than 10 years

Program/Application Information

RQ-2, RQ-3 The purpose of these questions is to correlate the size of the program, the nature of the system/program, the software type. This should offer insight into whether embedded and non-embedded use the same strategies and whether successful strategies are similar.

- 5. Is the system you are working on or reporting on embedded or non-embedded?
  - a. Embedded
  - b. Non-embedded
- 6. It is possible for a system to be embedded and the software to be non embedded (for example, database software may be part of a flight system but not itself embedded). Conversely, it is possible to work on embedded software for a non-embedded system (for example, embedded flight software components for a desktop simulator). Is the software you are working with or on or reporting on embedded or non-embedded?
  - a. Embedded
  - b. Non-embedded
  - c. Both
- 7. What type program are you working on (i.e. what is the final product)?
  - a. Satellite
  - b. Ground Station
  - c. Missile/Rocket
  - d. Helicopter
  - e. Submarine
  - f. Deep Space (probe or lander)
  - g. Logistics
  - h. Data Collection
  - i. Other (describe)
- 8. How large is the software effort on your program, or the program you are reporting on (approximately) (KSLOC)?
- 9. What type of application is the focus of the work product you are producing? i.e. graphics, GNC, algorithm, web-based, business, data mining, hardware components, architecture etc?

Reuse Information

RQ-4, This set of questions helps identify the type of reuse strategy employed, whether success is improved with being part of the decision, and whether the program is far enough along to measure factors that occur late in the program. We are able to compare development strategies and artifacts used on embedded systems vs. nonembedded systems.

- 10. Is your program employing product reuse? (artifacts, models, etc)
  - a. Yes (Branch to 11)
  - b. No
- 11. What approach to reuse did your program take? Check all that apply
  - a. Component based
  - b. Model based
  - c. Product Line
  - d. COTS/GOTS
  - e. Heritage/legacy
  - f. Ad Hoc (using already developed code that you happen to have around)
  - g. Other
- 12. Did you have input in the reuse decisions?
- 13. What phase has your program reached?
  - a. Capture
  - b. Requirements
  - c. Architecture
  - d. Design
  - e. Implementation
  - f. Integration and Test
  - g. Deployment
  - h. Maintenance/Operations and Maintenance
- 14. What product(s) is/are being reused? (i.e. requirements, architecture, models (what type?), use cases, code, drawings, hardware, test products, already tested clusters)
  - a. Requirements
  - b. Code
  - c. Architecture
  - d. Models
  - e. Drawings
  - f. Hardware
  - g. Use Cases
  - h. Test Products
  - i. Already tested clusters
  - j. Other (fill in)

Reuse Effectiveness Information

RQ-4 This set of questions will help correlate the effectiveness of the strategy against the strategy by identifying and scoring the change in outcomes attributed to reuse.

- 15. Did the reuse save labor hours?
  - a. No
  - b. Yes, from 10 to 20 per cent

  - c. Yes, from 20-30 per cent
  - d. Yes, more than 30 per cent
  - e. It cost us time (10-20 per cent)
  - f. It cost us time (more than 20 per cent)
  - i. * How much of the time savings to you attribute to reuse? Please explain how it saved time
  - i. * Please explain how the reuse cost you labor hours

- 16. Did you notice fewer defects than when reuse was not employed?
  - a. No
  - b. Yes, 0-10 per cent fewer
  - c. Yes, 10-30 per cent fewer
  - d. Yes, more than 30 per cent fewer
  - e. No, we observed more defects

- 17. Did it reduce testing labor hours?
  - a. No
  - b. Yes, 0-10 per cent reduction
  - c. Yes, 10-30 per cent reduction
  - d. Yes, more than 30 per cent reduction
  - e. No, we had to test more than 10 per cent more

- 18. Did it reduce items that needed to be tested?
  - a. No
  - b. Yes, 0-10 per cent reduction
  - c. Yes, 10-30 per cent reduction
  - d. Yes, more than 30 per cent reduction
  - e. No, we had to test more than 10 per cent more

- 19. Did you feel risk (cost, schedule, technical) was reduced? yes/no
- 20. * If you felt risk was reduced, please explain how. If not, please explain.

Appendix C

The MANOVA Tables

System Type vs Development Method

Dependent Variable	(I) a	(J) a	Mean Difference (I–J)	Std. Error	Sig.a	95% Confidence Interval for Difference	
						Lower Bound	Upper Bound
Component Based	0	1	–.215	.110	.055	–.435	.005
	1	0	.215	.110	.055	–.005	.435
Model Based	0	1	–.101	.101	.322	–.302	.101
	1	0	.101	.101	.322	–.101	.302
Product Line	0	1	–.142	.110	.201	–.360	.077
	1	0	.142	.110	.201	–.077	.360
COTS/GOTS	0	1	.086	.108	.429	–.129	.300
	1	0	–.086	.108	.429	–.300	.129
Ad Hoc/Legacy/ Heritage	0	1	–.037	.110	.738	–.256	.182
	1	0	.037	.110	.738	–.182	.256

Based on estimated marginal means
a Adjustment for multiple comparisons: Least Significant Difference (equivalent to no adjustments).

Multivariate Tests

	Value	F	Hypothesis df	Error df	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Powerb
Pillai's trace	.076	1.179a	5.000	72.000	.328	.076	5.894	.396
Wilks' lambda	.924	1.179a	5.000	72.000	.328	.076	5.894	.396
Hotelling's trace	.082	1.179a	5.000	72.000	.328	.076	5.894	.396
Roy's largest root	.082	1.179a	5.000	72.000	.328	.076	5.894	.396

Each F tests the multivariate effect of a. These tests are based on the linearly independent pairwise comparisons among the estimated marginal means.
a Exact statistic
b Computed using alpha =

Univariate Tests

Dependent Variable		Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Powera
Component Based	Contrast	.898	1	.898	3.798	.055	.048	3.798	.486
	Error	17.974	76	.236					
Model Based	Contrast	.198	1	.198	.993	.322	.013	.993	.166
	Error	15.148	76	.199					
Product Line	Contrast	.391	1	.391	1.665	.201	.021	1.665	.247
	Error	17.827	76	.235					
COTS/GOTS	Contrast	.143	1	.143	.631	.429	.008	.631	.123
	Error	17.191	76	.226					
Ad Hoc/Legacy/ Heritage	Contrast	.027	1	.027	.112	.738	.001	.112	.063
	Error	17.922	76	.236					

The F tests the effect of a. This test is based on the linearly independent pairwise comparisons among the estimated marginal means.
a Computed using alpha =

Figure C.1: MANOVA Tables System Type vs Development Approach

System Type vs Artifacts

Pairwise Comparisons

Dependent Variable	(I) a	(J) a	Mean Difference (I-J)	Std. Error	Sig.b	95% Confidence Interval for Differenceb	
						Lower Bound	Upper Bound
Requirements	0	1	-.194	.110	.081	-.412	.024
	1	0	.194	.110	.081	-.024	.412
Code	0	1	-.151	.093	.110	-.337	.035
	1	0	.151	.093	.110	-.035	.337
Architecture	0	1	-.221*	.110	.047	-.439	-.003
	1	0	.221*	.110	.047	.003	.439
Models	0	1	-.090	.111	.416	-.310	.130
	1	0	.090	.111	.416	-.130	.310
Drawings	0	1	-.030	.093	.744	-.215	.154
	1	0	.030	.093	.744	-.154	.215
Hardware	0	1	-.245*	.107	.026	-.458	-.031
	1	0	.245*	.107	.026	.031	.458
Use Cases	0	1	-.204*	.099	.043	-.401	-.006
	1	0	.204*	.099	.043	.006	.401
Test Products	0	1	-.269*	.107	.014	-.483	-.055
	1	0	.269*	.107	.014	.055	.483
Tested Clusters	0	1	-.098	.049	.052	-.196	.001
	1	0	.098	.049	.052	-.001	.196

Based on estimated marginal means
 * The mean difference is significant at the
 b Adjustment for multiple comparisons: Least Significant Difference (equivalent to no adjustments).

Multivariate Tests

	Value	F	Hypothesis df	Error df	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Powerb
Pillai's trace	.153	1.365a	9.000	68.000	.221	.153	12.289	.610
Wilks' lambda	.847	1.365a	9.000	68.000	.221	.153	12.289	.610
Hotelling's trace	.181	1.365a	9.000	68.000	.221	.153	12.289	.610
Roy's largest root	.181	1.365a	9.000	68.000	.221	.153	12.289	.610

Each F tests the multivariate effect of a. These tests are based on the linearly independent pairwise comparisons among the estimated marginal means.
 a Exact statistic
 b Computed using alpha =

Univariate Tests

Dependent Variable		Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Powera
Requirements	Contrast	.730	1	.730	3.131	.081	.040	3.131	.416
	Error	17.731	76	.233					
Code	Contrast	.443	1	.443	2.621	.110	.033	2.621	.359
	Error	12.852	76	.169					
Architecture	Contrast	.948	1	.948	4.065	.047	.051	4.065	.512
	Error	17.731	76	.233					
Models	Contrast	.159	1	.159	.668	.416	.009	.668	.127
	Error	18.059	76	.238					
Drawings	Contrast	.018	1	.018	.107	.744	.001	.107	.062
	Error	12.700	76	.167					
Hardware	Contrast	1.163	1	1.163	5.184	.026	.064	5.184	.613
	Error	17.055	76	.224					
Use Cases	Contrast	.807	1	.807	4.218	.043	.053	4.218	.527
	Error	14.539	76	.191					
Test Products	Contrast	1.407	1	1.407	6.269	.014	.076	6.269	.696
	Error	17.055	76	.224					
Tested Clusters	Contrast	.185	1	.185	3.897	.052	.049	3.897	.496
	Error	3.610	76	.047					

The F tests the effect of a. This test is based on the linearly independent pairwise comparisons among the estimated marginal means.
 a Computed using alpha =

Figure C.2: MANOVA Tables System Type vs Artifacts

Appendix D

Interview Letter

Candidate Email I am doing research for my PhD on reuse effectiveness. In particular, I am comparing reuse effectiveness between embedded and nonembedded systems, and among various strategies for reuse (and which strategies work best for which types of systems). I was wondering whether you would be willing to participate.

The interview will take about one hour. This is not a company affiliated study. Your participation and answers will be kept confidential, with the information aggregated and coded so it will not be possible to attribute comments to a particular individual.

I am looking at reuse strategies (i.e. product line, component-based, model-based, legacy/heritage, ad hoc, ontology and any other strategy you may have worked with as well as any combination of the strategies) and which artifacts are reused within these strategies (i.e. requirements, architecture, design, models, source code, test drivers and other test products, test clusters (items already tested together) and any other artifacts you reused). I would like to know the outcomes, positive and negative, for effort, quality, time to market and other variables you consider important.

If you agree, I will be sending you a consent form to sign before the interview as per University of Denver interview protocol. As I said, this is not a company affiliated research project, so you will be signing it as an individual rather than as a representative of the company. The company will be masked so nobody can identify it, however, of course you don,Äôt want to divulge any proprietary information. The interview will be recorded and

transcribed. You will be given the opportunity to review and edit your responses to remove anything that may be company proprietary and to ensure the accuracy of the transcription or to clear up anything you choose.

If you choose to participate, I thank you. If you would like a copy of the final paper, I will be happy to provide it to you.

Please respond to jfvarnellsarj@me.com. Thank you.

INFORMED CONSENT FORM

ATTACHMENT B

(Title of Research Project)

You are invited to participate in a study that will (purpose statement). In addition, this study is being conducted to fulfill the requirements of a class in (name of class). The study is conducted by (student name). Results will be used to (purpose) and to receive a grade in the course. (Student) can be reached at (phone/e-mail). This project is supervised by the course instructor, Dr. Anneliese Andrews, Department of Computer Science, University of Denver, Denver, CO 80208, (phone number, e-mail address).

Participation in this study should take about 60 minutes of your time. Participation will involve responding to (#) questions about (question content). Participation in this project is strictly voluntary. The risks associated with this project are minimal. If, however, you experience discomfort you may discontinue the interview at any time. We respect your right to choose not to answer any questions that may make you feel uncomfortable. Refusal to participate or withdrawal from participation will involve no penalty or loss of benefits to which you are otherwise entitled.

Your responses will be identified by code number only and will be kept separate from information that could identify you. This is done to protect the confidentiality of your responses. Only the researcher will have access to your individual data and any reports generated as a result of this study will use only group averages and paraphrased wording. However, should any information contained in this study be the subject of a court order or lawful subpoena, the University of Denver might not be able to avoid compliance with the order or subpoena. Although no questions in this interview address it, we are required by law to tell you that if information is revealed concerning suicide, homicide, or child abuse and neglect, it is required by law that this be reported to the proper authorities.

If you have any concerns or complaints about how you were treated during the interview, please contact Paul Olk, Chair, Institutional Review Board for the Protection of Human Subjects, at 303-871-4531, or you may email du-irb@du.edu, Office of Research and Sponsored Programs or call 303-871-4050 or write to either at the University of Denver, Office of Research and Sponsored Programs, 2199 S. University Blvd., Denver, CO 80208-2121.

You may keep this page for your records. Please sign the next page if you understand and agree to the above. If you do not understand any part of the above statement, please ask the researcher any questions you have.

I have read and understood the foregoing descriptions of the study called (name). I have asked for and received a satisfactory explanation of any language that I did not fully understand. I agree to participate in this study, and I understand that I may withdraw my consent at any time. I have received a copy of this consent form.

Signature _____ Date _____

(If appropriate, the following must be added.)

___ I agree to be audiotaped.
___ I do not agree to be audiotaped.
___ I agree to be videotaped.
___ I do not agree to videotaped.

Signature _____ Date _____

_____ I would like a summary of the results of this study to be mailed to me at the following postal or e-mail address:

_____

Figure D.1: Informed Consent Form

Appendix E

Interview Questions

Questions The following questions were created to guide the semistructured interviews. As interviews progress, some may be modified somewhat, others may not be asked, either due to time constraints or irrelevance in terms of the particular interview.

- What is your background in reuse? This question is meant to provide insight into the viewpoint of the subject. If the subject has worked on multiple projects involving reuse, it will allow the interviewers to pursue comparisons.
- What types of systems have you used reused products on - embedded vs non-embedded? If the subject has worked on both embedded and non-embedded systems, it will allow the interviewers to look for the differences and similarities. If the subject has primarily worked on one or the other, it will allow the interviewers to pursue more detail on the type the expert is most familiar with. [RQ2,3]
- What reuse strategies have you used? Why? What were the outcomes? Because the survey observed significant differences in the strategies used between embedded and non-embedded systems, this will allow the interviewer to validate the survey results and seek reasons for the difference.[RQ3,4,6]
- What artifacts have you reused? Because the survey observed significant differences in the artifacts used between embedded and non-embedded systems, this will allow the interviewer to validate the survey results and seek reasons for the difference.[RQ7]

- What was the level of reuse? This question should help the researchers determine whether reuse is used more or less given factors such as type, strategy and artifacts. [RQ1,2,4,5]
- Have you encountered any obstacles? This question is used to elicit the types of obstacles encountered by each type of reuse. It should hopefully show whether the obstacles are the same or different between embedded and non-embedded systems. [RQ1,2,4,5]
- How do you define success/failure? Success criteria may differ, this should give us an idea of the differences or similarities. [RQ1, 2, 5]
- How successful were you? This should help indicate whether there are differences in success levels between embedded and non-embedded systems. [RQ1, 2, 5]
- What drove the success or failure? This should give us the reasons for success or failure. [RQ1, 2, 5, 8]
- How do you define component? Model? If there is a difference in how embedded systems and non-embedded systems define these terms, there could be greater or fewer differences between the types than previously thought. [RQ4, 5]
- What were the benefits of reuse? Non-benefits? This should provide us reasons for any differences in reuse experiences. [RQ1, 2, 5]
- What about reuse and nonfunctional requirements? Do these types of requirements impact reusability? [RQ1, 2, 5]
- In your opinion does reusing the hardware make a difference? Why or why not? This could tell us the reason for reuse strategy and artifacts. [RQ5]

Appendix F

Finding Convergence in Interview Responses

The following spreadsheet was used to check for convergence. Responses are in boldface type the first time they are heard, in normal type face thereafter. Note that after twelve interviews, there are no more bold comments.

Person	A	B	C	D
What is your background in reuse?	35 years	7 years chief architect a few years as developer	22 years	8 years FPGA, 20 years software development
What types of systems have you used reused products on, embedded vs non-embedded?	Non	Non	Non	Emb
What reuse strategies have you used? Why?	**heritage/ legacy; model based**	**cut and run**; **one baseline across multiple programs;** Heritage; **Component**; **take an snapshot of someone else's code and then I would modify**; **Service Oriented Architecture**	**scavenging reus**e; **similar code or artifacts in similar projects and adapting for use**; I have never worked on a reuse project or program which provided me anything in reuse that I could actually adopt with low cost; design patterns; three attempts at repositories; they tout model-based development, reuse there, like everything else I've looked at it appears that is an advertisement not an adopted culturally sound process	much of the heritage is ad hoc;
What artifacts have you reused? Why?	**software applications; not so much models; database; data sets; software applications; models as in models and simulation; components; test drivers; test products; test clusters**	software code; **design artifacts, the interface specifications, requirements themselves; service level agreement; automated testing; architectures; COTS; infrastructure**	**reuse libraries; UML designs; Design patterns;** reused requirements	library for reuse; components; **interrupt controllers, clock domain processing, multiple resets, cross clock domains, DMA bus drivers;** auto testing; **requirements; verification methodology based on industry standard**s; **IPcore; performance models; you use a tool to convert it directly to the FPGA code**

Figure F.1: Expert Responses

Person	E	F	G	H
What is your background in reuse?	34 years	26 Years	25 years	16 years
What types of systems have you used reused products on, embedded vs non-embedded?	Emb	Non	Emb	Non
What reuse strategies have you used? Why?	**Libraries; designed for reuse; ontology**	Copying and pasting; **black box reuse**; Component Reuse; model based	**Product Line**; no strategy for creating reusable pieces; library; ontology	Adding capability into existing baseline
What artifacts have you reused? Why?	requirements specifications, design specifications, code reuse, the test cases, and test scripts; test clusters	code artifacts; **system models; reference architectures;** Requirements; test procedures	Code; Framework; **Utilities;** Requirements; Simulation framework; Design requirements; Architecture; test drivers; test products; test clusters; standards; algorighms; design; reference architecture	Data; **Documentation;** Requirements; Design; **Scenarios; FOSS**

Person	I	J	K	L
What is your background in reuse?	25 yrs	20 years	20 years	16 Years
What types of systems have you used reused products on, embedded vs non-embedded?	Non	Both	Emb	Emb
What reuse strategies have you used? Why?	Legacy; service oriented architectures	**mega reuse strategy; framework**	**opportunistic reuse**, I did project A, now I'm working on project B, I did something on A, I'm going to clone copy and reuse on B, that's kind of the lowest form that relies on just that the human knows that he's done something before, uses it for the next thing; Sometimes no modification, sometimes modification; infrastructure middleware code is basically reusing without change, but some stuff is actually copied and modified	**Model-based**
What artifacts have you reused? Why?	**scientific algorithms; workflow control;** COTS; test environment; requirements; Components; Data; Design; Interfaces; performance models	Requirements; Models; Architecture; Components	Code, **COTS, build infrastructure**, requirements, design, design documents, test documents, test, test products, **higher-level architecture components,** reference architecture to go into particular domains or disciplines	the UML stuff, requirements, diagram reuse

Person	M	N
What is your background in reuse?	30 years	28 years
What types of systems have you used reused products on, embedded vs non-embedded?	Emb	Emb
What reuse strategies have you used? Why?	Productized the system; **COTS/GOTS**	Legacy
What artifacts have you reused? Why?	reused the design, reusable, **rehostable operating system;** requirements, architecture, components, test products, test clusters; models as well simulation models, performance models, UML, documentation	code fragments, design and requirements artifacts a code reuse, documentation, environment, testbeds

Person	A	B	C	D
What were the outcomes?				
What was the level of reuse?	70%	**it is in some instances it is very specialized for a particular problem and the reuse is low; algorithm is 80-90%; our software baseline is not really reusable at all**	**project A: 65%. Project B: 20%, project C: 15%. Project D: 20%. Project E: 28%.**	
Have you encountered any obstacles?	**If the software has to be modified; don't always follow the same discipline and the same elements, the same structure; make it adaptable to the particular environment**	**Cultural; Not invented here; biggest barrier to reuse is the cut and run strategy; forking; reuse was way overstated; missed opportunities**	**No way to check that what was done actually represented reuse; fudging a lot of things to make numbers come out right over the years; how to produce a reusable artifact that stays useful?; bit rot, (anything that sits on the shelf becomes less useful over time. It's a decay function); understanding the business case is what's going to keep them up-to-date; capturing enough metadata, the time to do a make versus buy trade, whether that component is the one you want to invest in; how easy is it to try to reuse an artifact and then bail on it if my initial trial, does not prove satisfactory; the costs of any project are driven by the decisions you make in the first few months.**	**many senior engineers, tech leads, design managers, don't want anything to do with it; level of pushback from technical leads, principle engineers, and senior management; malicious compliance; the way we do architecture development; Once you start low level assembly language coding, you can't rearchitect, you're just basically patching; big ball of mud; lack of high level preparation, lack of documenting requirements or making sure requirements are comprehensive, complete, accurate, lack of testing it correctly; So that is actually probably the number one issue to low level coding is antiarchitecture**

Person	E	F	G	H
What were the outcomes?	Definitely mixed	we satisfy 92% of the requirements, right out of the box, I also have all the test plans done, all the test results done; I can show compliance and prove it		
What was the level of reuse?		15-20%	Some systems 60 to 70%; But other systems it's more like 20 to 30% reuse.	
Have you encountered any obstacles?	**they wouldn't use something they found unless they could satisfy themselves about its pedigree.; biggest obstacle is lack of design for reuse; second biggest obstacle is the not invented here syndrome**	**The obstacle to reuse is if you don't have that (chief architect in control) kind of setup then it's not going to happen; "Why would a software developer spend 10 minutes looking for something he could easily write in a day or two?"**	**we don't have an actual strategy for reuse from the business unit; younger software engineers view everything that previously exists as old-fashioned If they look at it and see one thing that they don't like, they're willing to pulse out the whole thing; anyone wants to start up a reuse workgroup or try to make a business unit reuse strategy, what they want to do is have a one size fits all. And that's impossible.**	account for it when we estimate effort for a job

Person	I	J	K	L
What were the outcomes?			It's rare that it's so bad that you wind up costing more, maybe 90, 95% of the time it doesn't cost more. But if failure is costing significantly more than what you thought, that probably happens not half the time, butabout 30, 35% of the time	
What was the level of reuse?	rule of thumb is if the software reuse is over 20% you need to be suspicious.; If you are reusing an existing scientific algorithm 80% reuse is I think reasonable		over 20% is rare	I can see reuse in the 80% or higher area
Have you encountered any obstacles?	**dependence upon so many vendors; We always underestimate our software costs; documenting**		**Misfit;** the people who quoted the reuse did not understand how much the design and the requirements, and then the code, would have to change; incorrectly reusing things that aren't quite right, **being too optimistic** thinking you didn't have to change anything, when you actually did have to change it; **customers tend to put clauses and constraints on the reuse between their programs and other programs**; a not invented here kind of attitude " I'd rather redo it myself than use someone else's;" forking of the products	what I call the human elements; , find that it's easier to build or they would rather build than seek somebody else's work and figure out how to morph it; a not invented here; if you don't plan to build it for reuse it becomes very difficult to reuse; **understanding someone else's code is harder than just doing it yourself**; when they fork they maintain the separate baselines rather than make them common

Person	M	N
What were the outcomes?	it works pretty well for us	Very high if just updating an existing system, much lower if omporting from IRAD
What was the level of reuse?	If we were able to use 75%, I would say we exceeded our expectations	
Have you encountered any obstacles?	In creating those models, we need **experts** to make sure that the model is doing what it's supposed to, but we also need newer people coming out of college who are experienced with modeling and the modeling languages and techniques and mesh these two together that the major obstacle goes back to that contractual restriction of not being able to use them across major programs	In my mind the biggest obstacle to software reuse is more cultural and people driven, not technical driven I have seen it fail many times when the code itself and all the other products was just flat out given to another team who, and there's not a real good **technology transfer** methodology, meaning in certain people under that team or collaborating with what needs to be done and stuff like that So that has seemed to fail many times and I don't know if it's because it was not invented here, or another is that there are technical incompatibilities also

Person	A	B	C	D
How do you define success/failure?	**Success if you can make it fit in your environment and it passes all the tests, unit testing, and if you can validate and verify it works in the new environment; Failure is software that it doesn't work picking it up and using it in your environment and you cannot modify it cheaper or faster than you could have developed it from scratch**	**satisfy the requirements within cost and schedule constraints**; **build a reusable solution and to build the artifacts around it**; failure, you don't satisfy your requirements and your system doesn't execute as expected; **you fork the baseline; if you save money at the beginning only to cost more money at the end**	**if I can deliver my product and have either shortened the delivery time, or had a reduction in the number of staff needed for that particular function, then I've been successful**; **Failure is having to select an alternative, having to invest my own additional resources,** that would exceed what it would've taken me to do the job from scratch	Did it save time and schedule by doing reuse? Defects; Failure: **some cases it took longer to reuse code because of all the issues with it,** it would have been to just do it from scratch, so did it save you time and schedule?; Sometimes you don't find out until integration and test at a subsystem level and all of the sudden you start having issues; And then **you find out the code wasn't as good as you thought it was**, and particularly when software tries to interface with it it has some glitches in it, unidentified bugs in it, so sometimes it you have to wait until the end of the program to judge whether or not it was a success for reuse
How successful were you?.	**more common that it doesn't work**		**I've suffered two significant failures. But the number of successes are between five and 10 times that number**	I did a survey of the things that cause the most issues for people. And those things, everyone used them and not a single person had a bug in any of those circuits, so I eliminated some of the major sources of bugs; So that's one point where we had a **pretty good return on investment, and that was demonstrable**

Person	E	F	G	H
How do you define success/failure?	success if you decrease the problems over what you would have had in the way of cost and schedule; the perhaps more sophisticated measure would be if you actually **improve product quality without any significant increase in cost and schedule** by trying to do the reuse;; failures of expectations; we have made a commitment to customers in terms of cost and schedule based on what we believed we could achieve with reuse and the **reuse didn't pan out as planned and therefore we had cost and schedule above the commitments to the customer**	count up your per cent reuse; , we saved the program 6 million bucks.	I guess it's a success when you can reuse; I guess a failure would be, pull a piece in and it was totally not applicable to your job.; If I was able to reuse the code, but it cost me twice as much as I anticipated, than that would have to be viewed as a failure; So if we **underestimated,** that's where I think the failure comes in, where you fail to estimate the actual cost of doing that particular module.	success is probably related to cost and schedule.;; Failure we just didn't get the level of reuse that we had hoped out of this other piece.
How successful were you?.	**very slightly successful**	**I haven't failed yet on big projects**	We've been **able to generate a lot of new systems**, and because these three programs were using similar electrical designs and similar missions and similar requirements and similar software frameworks we were able to pull the necessary pieces to create an entirely new system, field it and fly it successfully in six months.	**Most cases have been pretty successful.**

Person	I	J	K	L
How do you define success/failure?	cost and schedule		success is that in the end the reuse actually costs less than it would have cost to rewrite it, and have a high percentage of the performance you would have gotten if you had written it from scratch; it costs less, and is performing near what would be if you designed and wrote it; Failure is that you didn't know what was needed so you're taking a **sub optimal design for the problem**. People rewrite because it's actually a better **fit;** people are over optimistic before requirements decomposition from the higher level results in a set of requirements from the systems engineering that's a mismatch	Success - if an individual is willing to take the time to look for and find the component that can be reused, knowing that in all likelihood, that will save time in the long run; failure occurs when **they give up too soon**, and don't allow the time to do that and recognize that the payoffs can be quite beneficial
How successful were you?.	It was a nightmare partially because it was our first full implementation of the service oriented architecture.; In the most recent delivery, which we are getting ready to promote, has gone remarkably well, and it's because we have learned lessons about how to properly structure integration and test.			

Person	M	N
How do you define success/failure?	Success: To be able to reuse software artifacts, and I'm going to say, as expected or planned; Failure: If we were not able to reuse the amount of software that we thought we would, and as a result cause cost or schedule of the project to fail, in other words to not be able to meet cost of schedule	Success is, contractually, if you go into a bid baseline and you say I'm going to bid some 40% reuse based on some technical assessment you did on where you would be reusing code, and then you would achieve significantly less than that, then I would say was a failure; most of the failures are non-technical, it might be a technically solid portion of code that really works fine, but then when you take that little piece of code and put it into a larger architecture that has different coding and other standards, there could be a technical incompatibility, but it's typically driven by people making decisions about deciding not to reuse that code for one of those reasons
How successful were you?.		

Person	A	B	C	D
What drove the success or failure?	**So success is determined by how similar the use is; not designed to be reused; if it is very dissimilar you'd better be careful**		Well, I'd like to think experience; I've reduced my expectations.	
How do you define component?	**it depends on what you are talking about**, a **component supports the subsystem capability**, in other words you have a capability within a software set that will fit on that component and the purpose of that component	**component is essentially equivalent to a service and a service is really just a piece of software and perhaps the underlying hardware** that **performs a particular discrete function with a well defined interface and either a service level expression or a set of requirements that describe what that service is**	**an artifact with a well-defined inputs, outputs and behavior**.	Component to me, speaking from the hardware end, is **a physical entity**, such as the FPGA is a component. At a deliverable level, the assembly code is included. I do not use the term component based engineering because between the hardware and software it is confusing

Person	E	F	G	H
What drove the success or failure?	Well, the biggest thing, I think is the **ability to correctly forecast future demand**; The biggest failure factor is the **assumption of reusability without making any investment in reusability upfront, or be checking to see if there's a match.**	What made it successful was **having the processes and the management and everybody participating at the front**	There are **reference architectures** for these product lines, and you can do reuse along with the reference architectures, **but doing reuse without taking those into consideration is a failure**.	Success is the degree to which we really **follow our guidance for evaluating the reuse that we are going to inherit; expertise**
How do you define component?	**a software unit to a C SCI**; if you're **reusing the hardware and software as a unit, well obviously you could call that a system complement, or hardware component**	software components - **a piece of code with its associated documentation that you use without changing it.; We did talk about test cases as a type of component**.	component would be the **CSC,** which is basically something in your system that performs a function **given to a single developer and is decoupled from the system**, it's a place where you can minimize the couplings to the system.; **Hardware is independent**.	Component, I think, for me, is I tend to think of it more along the lines of a class or a capability, some strongly cohesive, loosely coupled thing

Person	I	J	K	L
What drove the success or failure?	**Our ability to manage integration and test to find all of the defects before we go to site acceptance test, I think people around here would say that's been our greatest contributor to recent success.**		people willing to really consider the reuse and look for it and be open to taking something that maybe they don't think is the optimal or the best design or implementation, but it will work understand existing code, obviously, and software, and requirements and how its implemented in the architecture, that kind of stuff, you have to know that fairly well	
How do you define component?			component is **a term I usually don't use because it is so vague**. a component is **a thing I want to try to reuse**, it tends to be a set of **implementation classes** it might be four, eight, 16, maybe, that could come together to do a useful function; sometimes your component's hardware and software together There are situations where we have a **set of multiple classes on a set of hardware and it's actually a component if we put it in the system to reuse**	A component is an individually discernible minimalist thing that does a particular function. a component to me is a small thing that can be aggregated to the larger; It's where you sit and how you are drawing your boundary; at the lowest level of detail components get aggregated together and as those aggregations get larger and larger the ability to juggle those individual components mentally requires some abstraction

Person	M	N
What drove the success or failure?	having it within a product line was a big one	Where I have seen success it's almost always been same team leveraging what they have written before. If the people have written the code, or understood it, I've seen that reuse work very often. I have seen it fail many times when the code itself and all the other products was just flat out given to another team and there's not a real good technology transfer methodology. On reason is not invented here, another is technical incompatibilities
How do you define component?	A software component, in my experience, can be anything from a software module all the way up to a CSCI. If you are talking about a subsystem component or system component, yes, it includes the hardware. If you are talking about a software component, no	I'm used to decomposing the software item for the embedded into loosely weaved, often grouped into domains, like a particular set of cohesive functionality tightly coupled technically with each other but loosely coupled to the rest of it, and within that domain you have software components which are a group of functionality tightly coupled within that component but loosely coupled with some other components associated with it.

Person	A	B	C	D
How do you define Model?	**Models can be like models and simulation, using physics to emulate, or simulate the actual hardware, functioning of hardware and the software and making it work, a simulation of the hardware that you could run the software against; And then, you go into what we call model based engineering, where you find a model ia totally different - a model would be in Rhapsody and could use SySML or UML and all the artifacts that are associated with that, it could include activity diagrams, data flow diagrams, needlines; It has multiple meanings, and it depends on how you use the word**	we mostly focus on the **performance model**; focused on the performance rather than necessarily the functional performance of the system; **predominantly latency and throughput; System bandwidth,** and really understanding where the performance bottlenecks are; If we were starting from scratch, especially with modern techniques, we would probably do a lot more model based architecture and design, but right now, most of the significant changes are understood changes to an understood system	a capture of either a **physical thing or a set of components that are already constructed to interact a certain way**, models are also I suppose, thought of as **hierarchical; Design patterns and design artifacts**	but we know the tools that enable SySML, Simulink, where you develop a model of the device, either as you are trying to develop the lower level entities or as you develop your algorithm application; Modeling and simulation fall under model based engineering.
What were the benefits of reuse?	if it is successful you **save money and time**; So you are saving your resources, you are saving your funding	its in not only the **development costs**, but the **integration and test savings** and the **repeatability; nightly builds**, **nightly integration testing**, and that has actually been a **cultural benefit to the quality of life for the software developer**	the efforts at **making easy to find things that will save me time and money are good**; I think of the different forms of reuse, that design patterns are one of the most promising	Well the benefits for reuse, that is for good reuse, is again, you can save a lot of effort and a lot of time, **it can produce a better circuits** as far as what I do at the low level; Its **easier on the designer** because they don't have to worry about how to implement some of the tougher parts of their design. They can concentrate on what they need to.

Person	E	F	G	H
How do you define Model?	there has not been one uniform definition of model, model-based systems engineering is a basis for software reuse; a model is a way to capture the essence of a system and/or software design so it can be viewed to get insights into different aspects of the design model; performance model such as Matlab and Simulink; a fully model-based development includes models of the system and software development, models of the environment and aspects of environment that are pertinent; for embedded systems development models or emulation models are critical; Having one monolithic definition of model-based makes no sense, implementation models, which emulation and simulation of systems; system models, and models of the environment.	models are a representation of reality, where you mean the system, so anything based on that is a model,; document the architecture this way using views, and each view has a certain type of diagram or diagrams to capture what you want and those are in fact the models	a model is a thing in the simulation that encapsulates some part of the system that is not in software. So you can encapsulate environmental things, physical things like wind, you can encapsulate a device. A sensor.	Model, for us, then, would be essentially how those (components)interact.
What were the benefits of reuse?	probably the biggest benefit is up **at the meta level**. That is it **focuses the business attention on what you have to do to get value out of the engineering that you're doing or how you might get more value out of the engineering you're doing**.		**affordability,** that's been the primary benefit. The other benefit is the ability to **accelerate schedule**; we reuse the patterns more they get refined,; It **reduces the risk** from building something else.; It's the **knowledge that this thing has worked over time**.; I would say it's **less defects** for a piece of reuse.	Well for me the benefit is always **productivity, potentially reducing complexity**

Person	I	J	K	L
How do you define Model?	One model would be this thing I call the template, which is **our structure for the way we are going to do the development** all the way from the time we are defining our requirements to the time we deliver to the customer.; The other model that we worry about is like the discrete event simulation types of model.		We have lots of different models. A model is essentially the architecture and the design artifacts; Simulations are models, we have system architecture models, static performance models, a whole set of different models, and those do fall within my definition, but specific to just software, when I talk about a software model, I'm talking about UML models of the architecture, design and implementation; something that's model based, the model-based engineering approach, I think incorporates all those together in an integrated set of models It would be your design model, your performance models, your simulation models, your architecture models, it's everything linked together	a model is probably an aggregation of multiple things. If the thing is so simple it can be modeled with one rectangle or, how should I say it? I think it's the size
What were the benefits of reuse?			if it works, it's reducing your cost, it's reducing your schedule and it's reducing your technical risk	Well I think the benefits are, the obvious one everybody goes to, there's a tremendous cost savings in the reuse and a big benefit if you can find the thing, and we use that phrase system boundary that defines what it is and have a good understanding of what it does and how it can be used and how it can benefit you and your larger system

Person	M	N
How do you define Model?	software simulation or software emulation of some system or subsystem	I'm trying not to use the term model anymore because I think it's extremely overused; I've seen model used many ways. It's some kind of representation of the real thing that could be a simulation model or an abstract representation of the software; I think the word model works very well when you put it in light of a simulation of something that's real, I'm going to model this from a simulation, but when you talk about your embedded end product and you start saying it's a model you ask will it elicit the real thing or is it a model because people think of a model is something I've built on my desk, a scale model of something that's real and so it's really a representation of it
What were the benefits of reuse?	one of the main benefits is cost, but we also get quality and schedule, because if we are using about 90% of the build and we only need to change about 10% of the build for this new version then we are getting all of that schedule and cost reduction and we're not creating new defects in the other 90%	it saves redoing the same thing just a different way. you could save the time, which is not only developing it, but debugging it and working out all the latent defects

Person	A	B	C	D
What were the Non-benefits?	If the reuse is **not going to meet your performance requirements**, then you have to evaluate what do I have to do to that reused software so it will meet my performance requirements, and **if that means I have to go in and redesign it and rebuild it then it may not be worth the reuse**	**collateral aspects of unintended consequences of reuse**; The more people that you have reusing a component the **less freedom of movement** you have for any individual stakeholder in that reuse	When it came in mandated to me, it was going to cost money whether I reused it or not, in fact the repository was not in good enough shape at that time to benefit more often than simply have to prepare a report that would **take me some number of hours of a senior engineer's time. So it ended up being a tax**	
What about reuse and nonfunctional requirements? Do these types of requirements impact reusability?	**it could make it easier if it is already met IA requirements, for instance,**	**the entire system is behind the curve on that front; security requirements, that's a prolific problem right now. DNI has instituted a different security accreditation methodology and that's filtering through all of the programs and so any reuse model, a component reused across multiple programs and the security requirements around that component should be sold off by definition, there should be a level of trust there that I don't have to resell those security requirements across the enterprise over and over and over again, but that's not where we are**	the **challenges you have capturing metadata about reuse artifact**s those are some of the more critical ones; and often, when you're trying to match a component to your needs, you've exceeded some design parameter of the reuse artifact producer and that is what you end up needing to evaluate to determine whether you can adapt or adopt it; building upon reusable patterns will make that problem tractable especially in the face of potentially changing regulations or standards	I think reuse particularly code that has been through a development that included some DO specs, **some security specs is very important because you don't want to keep doing the same thing wrong over and over again.**

Person	E	F	G	H
What were the Non-benefits?	I guess the only disadvantages I've seen are when business units or companies **undertake reuse without understanding what it takes to be reused successfully and then encounter business failure**,.			**sometimes when you pull in something, you pull in additional things and features you don't need**, which can be extra work to go and tie off these functions.
What about reuse and nonfunctional requirements? Do these types of requirements impact reusability?	**nonfunctional requirements reuse was as useful or more so, so things like robustness requirements.; the level of the security risk rose over time, so that systems we developed in the early 1990s, for instance, by the time we were reusing elements or entire systems of those by the late 1990s the security risk had gotten way up**	**the non-functional requirements are probably on a system by system basis**	Yes. I think the **expense of modifying for the ilities** and stuff like that a lot of times they relate directly to how the architecture was done.; There's not a lot of ilities the flow directly within a single component only, normally the ilities flow through the architecture.	All those things, we've kind of had an ongoing discussion about how do we handle security.

Person	I	J	K	L
What were the Non-benefits?			it **typically is not a perfect fit**; It **won't give the best performance or the most eloquent design** for that particular situation so you are often, I don't want to say shoehorning, but you are often fitting it into something that is not quite a perfect fit; That affects performance aspects, and **it increases risk**	The downside is how do you **qu**
What about reuse and nonfunctional requirements? Do these types of requirements impact reusability?	So with nonfunctional requirements, we're always worried about **how we're going to sell those off** and there is always a certain amount of negotiation up front about how we are going to show that we've met those requirements.; I'm always concerned about the **reliability** requirement. The reliability requirements really are mathematical models. **generally speaking, reuse helps us meet the nonfunctional requirements**.		Especially security requirements **are actually a detriment to reuse;** makes it so sometimes you would have to reject some products	

Person	M	N
What were the Non-benefits?	The non benefit is you inherit the software irregularities or inconsistencies, if it's got defects. But you also inherit the software and the artifacts of **which your team doesn't have any experience**	
What about reuse and nonfunctional requirements? Do these types of requirements impact reusability?	Reusing the requirements or the requirements statements is a common thing that we do on a program, so the nonfunctional requirements can actually cause you to have to either not reuse, or modify more than you'd like; We have some security issues that, depending on where that CSCI is going to operate, can restrict you from any reuse Because the security restrictions of that new product may need, while part of satisfying the security and the safety concerns comes from how you developed that piece of software	well I'm on the fence with that, because I'm not sure that all those, in all the programs that I've worked those have typically been levied by design, so I haven't seen someone explicitly take that requirement for security or reliability or whatever and actually map it to a feature in their code

Person	A	B	C	D
In your opinion does reusing the hardware make a difference? Why or why not?	Yes, because **if you know it is already compatible with the particular server or whatever the hardware is, then it is easier to do because you have already worked out any problems**	**yeah, absolutely.** The whole transition to commodity hardware has been liberating in our domain because we develop content for our system. We take content from other providers, so **standardization of the hardware architectures has leveled the playing field** and made it so when we get an algorithm from a lab or another contractor **there's much less development work now to integrate that into our system and harden it for operation; Platform independence - yes I'm a zealot in that particular argument**	certainly the kind of machine architecture your using is going to influence what is and isn't reusable; my ability to actually take this application and try it out gave me a very different picture of **sizing of that application on the directed hardware** so my choices were to say gosh I need to either invest in an adapting in that application or I need to re-specify the hardware; **The problem with platform independence is the same problem with what we call portability**. If you are trying to write portability from one device to another device, it may not work because each device has its own set of differences. **We're never going to get there and we may not want to**. **Because you won't get the performance out of the specific device**.	Reusing well documented, well designed, well tested hardware is an **extraordinary method of not only preventing errors later on but for affordability concepts**, everyone's pushing affordability; **And I think that reuse and commonality go together.**

Person	E	F	G	H
In your opinion does reusing the hardware make a difference? Why or why not?	**well, for embedded systems, it makes a huge difference**. Especially closest to the hardware platform independence may be more costly than it's worth; For other things, algorithms, for instance, it probably pays off enough that it's well worth developing it in a platform-independent way	Platform independence Its what we do	Well, certainly. If you are reusing the same hardware, there's a lot of pieces that you can reuse but in some sense we try to abstract away from the hardware a lot both to give us the capability of running in a simulation environment and for being able to port it,; because right now what we have in the embedded world especially is were having a lot of flux in the processor world.; But in the embedded world, we have all kinds of DSPs and devices and Intel for higher end embedded stuff is becoming much more popular but for the low powered stuff like tools, Intel is still a hard fit.	**We've had issues with platform independence and how independent everything can really be**.; There are things that we inherited, that we hadn't touched, we hadn't intended for there to be any design change to them, however they broke with the move to Linux..

Person	I	J	K	L
In your opinion does reusing the hardware make a difference? Why or why not?	Yeah it does. One of the problems we ran into in the past with scientific algorithms is if we move from one type of the processor to another, all the **underlying mathematical libraries changed**. And so the porting cost was high.		**in a well architected and designed software, it's less of an impac**t; Often we are changing the hardware without changing the software and as long as the software was designed in a way to be portable for the hardware and you have used isolation layers, it's not that big of a deal; Obviously you make it easier if you are using the same software and hardware on a program, but what we find is hardware moves way too fast these days, and as soon as you are reusing the software, the hardware it is running on is already obsolete; Going forward I think that that is something that we are not counting on, we are not counting on having the same hardware there, we are designing the software architecture to be tolerant of it changing	

Person	M	N
In your opinion does reusing the hardware make a difference? Why or why not?	For us, it would, because if we've got software working on a particular computer, and its hardware, if we can reuse that same computer CPU, board, whatever you want to call it, in another system, we can reuse the operating system, we can reuse bus interface handling software, things like that, that otherwise we might have to go develop. Platform independence we don't bother much with that, because of the specific systems that we are creating. I don't think platform independence would drive it any	Significantly, yeah. I believe that both the reuse of the processes that it runs on, and whatever avionics systems that you are connected to through the I/O paths, to the sensors, the actuators, whatever happens to be there, everything that taps into what flight software needs to do, the reuse of that would significantly drive the benefit of reuse of the software that already supports that hardware platform; the stuff in the middle, infrastructure code, executive, whatever you call it, is independent of the platform and independent of the application; But below that architecture, below that element you would have to have platform dependent modules that dealt with the particular type of platform.