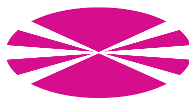
ON THE DEVELOPMENT LIFE CYCLE OF DISTRIBUTED FUNCTIONAL APPLICATIONS: a case study

PhD Thesis

LAURA M. CASTRO SOUTO



UNIVERSIDADE DA CORUÑA

2010

ON THE DEVELOPMENT LIFE CYCLE OF DISTRIBUTED FUNCTIONAL APPLICATIONS: a case study

PhD Thesis by

LAURA M. CASTRO SOUTO

supervised by

VÍCTOR M. GULÍAS FERNÁNDEZ

UNIVERSIDADE DA CORUÑA

2010

ON THE DEVELOPMENT LIFE CYCLE OF DISTRIBUTED FUNCTIONAL APPLICATIONS: a case study

a thesis submitted to / tese presentada na

UNIVERSIDADE DA CORUÑA

in partial fulfilment of the requirements for the Degree of
/ para a obtención do Grao de

DOCTOR/DOUTOR

Examining Committee / Tribunal:

Dr. Roberto Moreno Díaz (pres.)

Dr. José Luis Freire Nistal (sec.)

Dr. Thomas Arts

Dr. Ernst L. Leiss

Dr. Simon J. Thompson

A Coruña, 2010

A mi madre

Acknowledgements

I would never dare to say that I have walked all the way here alone. There are far too many people I owe a mention to, not only for their support during the months I devoted to writing this thesis, but most importantly, for their presence during the years before. Still, I want to make some special references.

> Xabier, for still being there, after so many years.
>
> Víctor, for his always wise advice and unconditional encouragement. Thomas, for his contagious enthusiasm and enlightening chat.
>
> Carlos, for his prompt disposition to help and shared teaching hours. My closest friends and family, for bringing me laughter and drifting my mind away. My lab colleagues, for making that a job does not feel like one.

Thank you very much.

Abstract

In a world where technology plays a major, increasing role day after day, efforts devoted to develop better software are never too much. Both industry and academia are well aware of this, and keep on working to face the new problems and challenges that arise, more efficiently and effectively each time. Companies show their interest in cutting-edge methods, techniques, and tools, especially when they are backed up with empirical results that show practical benefits. On the other hand, academia is more than ever aware of real-world problems, and it is succeeding in connecting its research efforts to actual case studies.

This thesis follows the mentioned trend, as it presents a study on software applications development based on a real case. As its main novelty and contribution, the integral process of software development is addressed from the functional paradigm point of view. In contrast with the traditional imperative paradigm, the functional paradigm represents not only a different way of developing applications, but also a distinct manner of thinking about software itself. This work goes through the characteristics and properties that functional technology gives to both software and its development process, from the early analysis and design development phases, up to the final and no less critical verification and validation stages. In particular, the strengths and opportunities that emerge in the broad field of testing, thanks to the use of the functional paradigm, are explored in depth.

From the analysis of this process being put into practise in a real software development experience, we draw conclusions about the convenience of applying a functional approach to complex domains. At the same time, we extract a reusable engineering methodology to do so.

Contents

Page

1 Introduction **3**
1.1 Motivation . . . 4
1.2 Objectives . . . 6
1.3 Structure and contents . . . 6

2 State of the art **9**
2.1 Software engineering . . . 9
2.1.1 Software development methodologies . . . 11
2.1.2 Software quality approaches . . . 18
2.1.3 Open challenges . . . 20
2.2 Imperative paradigm vs. Declarative paradigm . . . 21
2.3 Software testing: verification and validation . . . 23
2.3.1 Unsuccessfully addressed problems . . . 23

3 Case study **25**
3.1 Risk management . . . 25
3.2 Risk Management Information Systems . . . 26
3.2.1 Commercial software for risk management . . . 27
3.2.2 Prospectives . . . 29
3.3 The ARMISTICE project . . . 30
3.3.1 Data collection and analysis . . . 31

4 From requirements to analysis and design **33**
4.1 Requirements elicitation . . . 34
4.2 System analysis . . . 35
4.3 Program design . . . 35
4.4 Formalisation and model review . . . 36
4.5 ARMISTICE's business domain study . . . 37
4.5.1 Knowledge extraction . . . 38
4.5.2 Domain analysis and design . . . 39
4.5.3 System formalisation . . . 44
4.6 Formalisation benefits: validation planning . . . 50

5 Implementation of a paradigm shift **53**
5.1 Is such a change feasible? 54
5.2 Translation of borrowed concepts 55
5.2.1 Object-orientation in non-object-oriented environments . 55
5.2.2 Object-orientation and functional environments 57
5.3 Strengths of declarative languages 58
5.3.1 Functional patterns 59
5.3.2 Erlang 59
5.4 Developing key aspects of ARMISTICE 64
5.4.1 System architecture and technologies 64
5.4.2 Formulae and restriction language 66
5.4.3 Policy relevance 69
5.4.4 ARMISTICE as decision support system 71

6 Ensuring functionality and quality through testing **75**
6.1 Software testing 76
6.1.1 Verification versus validation 77
6.1.2 Testing levels 78
6.1.3 Testing techniques 80
6.2 QuickCheck 88
6.2.1 Property-based testing 90
6.2.2 State machine testing 92
6.3 Testing ARMISTICE 97
6.3.1 Data types verification 98
6.3.2 Integration testing 114
6.3.3 Business rules validation 129

7 Functional software development methodology **147**
7.1 Purpose 147
7.2 Phases 148
7.2.1 Requirements analysis and system design 149
7.2.2 Implementation 151
7.2.3 Verification and validation 152
7.3 SWOT analysis 155
7.4 Case study evaluation 157

8 Conclusions **161**
8.1 Contributions 161
8.2 Lessons learnt 163
8.3 Prospects: open research lines 165

Index **168**

Bibliography **169**

Figures

Figure Page

2.1 Waterfall software life cycle 12
2.2 Waterfall with prototyping software life cycle 12
2.3 V software life cycle 13
2.4 Prototyping software life cycle 14
2.5 Incremental software life cycle 14
2.6 Iterative software life cycle 15
2.7 Spiral software life cycle 16
2.8 RAD software life cycle 16
2.9 Extreme programming software life cycle 17
2.10 Unified Process software life cycle 18

4.1 Knowledge elicitation and validation process 38
4.2 UML model of risk groups and risk situations 39
4.3 UML model of risk situation dynamism 40
4.4 UML model of risk hazards 41
4.5 UML model of exposures to risk 41
4.6 UML model of insurance policies 42
4.7 UML model of accident claims 44
4.8 Versions and revisions of a risk situation 47
4.9 Correspondence between development stages and validation activities 51

5.1 ARMISTICE architecture overview 65
5.2 Policy clauses representation 72
5.3 Architecture of an AI production system 73

6.1 QuickCheck state machine 94
6.2 Decimal data type creation 100
6.3 Model-View-Controller architectural pattern. 115
6.4 Layers architectural pattern. 115
6.5 Risk groups and risk objects management use cases. 117
6.6 Role specialisation of selected management use cases. 120
6.7 Usage of a dummy component for integration testing 122

6.8 Layered application architecture sample 130
6.9 ER diagram example 132

7.1 SWOT Analysis 156
7.2 Real evolution of business objects in ARMISTICE 158

Publications

Page

3.1 Commercial Risk Management Information Systems. 28

7.1 Publications related to system design. 150
7.2 Publications related to model review and formalisation. 151
7.3 Publications related to system implementation. 152
7.4 Publications related to verification and validation. 153
7.5 Publications related to verification and validation (ii). 154
7.6 Publications related to verification and validation (iii). 155

1 Introduction

Software is everywhere. From tiny devices to huge equipment, from trivial purposes to critical tasks, software has become an essential part of many of the tools, services, and systems we use every day. It is in mobile phones and music players, it is part of coffee makers and television sets, it is present in cars and airplanes. Software is applied to as many different things as different dimensions our life has: buying tickets to a show or concert, making a doctor appointment, or filing tax. The same way electricity and the light bulb are considered one the most relevant improvements to human life in the XIX century, computers and software are undoubtedly changing our lifestyle at the dawn of this new millennium.

The process of building software in a structured, well-specified and repeatable manner is commonly known as *software development*, and carefully studied by *software engineering*. The term *software engineering* was born in the 1960s [1], with the appearance of the first high-level programming languages and the concept of reusability, which started to lay the foundations of the field. Ever since then, issues such as productivity and quality have been key aspects software engineers all over the world need to deal with.

A few decades after the so-called *software crisis* [2, 3] that, in its early days, put the discipline to the test through the 1970s and 1980s, we know now that there is no silver bullet in software development. At least, not just *one* single silver bullet. Many different instruments, technologies, and approaches have come and gone in a continual effort to solve all the open problems of software development once and for all. Some of them have even had quite resounding

success, but none has turned out to be the holy grail. Anyhow, as engineers, we have learnt something from every step of the way, and have continued looking for solutions.

This dissertation is an attempt to open a new line on software development, based on a new challenge and an old but improved paradigm. The challenge is the increasingly complex problems we need to solve, which are indeed only the result of the non-stoppable spread of information technology; their more and more demanding requirements force us to grant properties that were not even considered a few decades ago, such as interoperability, distributability, scalability, reliability, etc. The paradigm is *functional programming*, which has remained outside the industry, primarily constrained to academia and research, despite its particular features suggesting it as a powerful ally in the present situation.

1.1 Motivation

Soon after the first high-level programming languages were created, the concept of *programming paradigm* also came to light. It was the abstraction of the set of characteristics that made different groups of programming languages alike, on the basis of which aspects of computer programming they were focused on or built around. The most popular programming paradigms are probably *imperative*, *declarative*, and *parallel computing* paradigms, each of them presenting its own strengths and weaknesses.

Imperative programming languages, and among them procedural and object-oriented programming languages, have been possibly the most commonly used programming languages in the history of software programming, and they are still present in a vast majority of software developments nowadays. Imperative programming usually describes computation as a sequence of *statements* which affect the *state* of the program; since a statement evaluation may depend on the current state, the imperative paradigm lacks of *referential integrity* (also referred to as *referential transparency*), meaning that the same statement may have different effects (i.e., results) depending on where and when it is executed. One of the first and direct consequences of referential opaqueness is that automatic code optimisation (by pre-processors and compilers) turns into a much more complicated task. On the other hand, imperative languages are claimed to follow the closest philosophy to how the underlying hardware actually behaves, thus allowing for more efficient approaches in practise [4].

The declarative programming paradigm focuses on how a problem is described rather than on the actual way it is solved, thus shifting the abstraction level at which developers need to think. In particular, functional programming, a specific kind of declarative programming, is based on *mathematical functions* instead of statements, and avoids state and side effects. The foundations of

this paradigm are the lambda calculus [5], and even though in practise very few programming languages ensure referential transparency completely, it is still possible to have automated tools for behaviour prediction, verification, and code enhancement, among others. In contrast, declarative languages are often perceived as less efficient in the use of hardware resources, even though they usually present features such as lazy evaluation, which can potentially increase their performance.

Recently, improvements in hardware production have led to the introduction of parallel programming, a new paradigm whose philosophy is based on problem fragmentation and algorithm parallelisation. Of course, it is possible to approach parallelism from both an imperative and a declarative perspective, but mainly when we consider task-level or data-level parallelism. In recent years, nonetheless, fast innovation and popularisation of multi-core processors has lowered parallelism down to the *instruction level*. This has brought into the picture a whole new set of issues, ranging from synchronisation and communication problems to critical aspects inherent to concurrency (such as mutual exclusions and race conditions).

Most of the regular software developments do not require, however, fine-grain control over parallel hardware, so they can do without all the extra complexity. But why does industry seem to have forgotten about declarative/functional programming over the years? Solving more and more complex problems every day, we could take advantage of using the most high-level tools, that could help us focus on *what* we need to achieve rather than on *how* we are going to achieve it, not distracting us from the essence of the problem we want to address. Instead, the most popular development strategies and methodologies have completely omitted anything else but imperative languages. Standardisation and popularisation of object-oriented analysis and design techniques have only helped to consolidate that status quo [6].

But if we could combine the experience and know-how of broadly known and used software development tools and life cycles, with the benefits of functional programming, the improvement of both software production and software usage experiences could be enormous. And this is, precisely, the main objective of this thesis: to show that it is perfectly possible to accomplish a complete real-world software development from a declarative perspective. What is more, we aim to demonstrate that, by adopting this approach, considerable benefits can be obtained thanks to the properties of functional code. Explaining how a regular software development process can be put into practise with a functional orientation in the same repeatable, structured, and methodological way as traditional software developments, we hope to make it more accessible to companies and developers, giving it a chance of succeeding outside the academic world. Showing how a functional development can contribute to software quality assurance, and to increment control and knowledge not only about the development, but also about the business process itself, helping to

protect both products and activities from well-know failure (or weak) points, we hope to make the strongest point for it.

1.2 Objectives

The main objective of this thesis is, first of all, to examine the traditional development of big and complex software systems, with the intention of analysing the whole process and identifying those stages which are more critical or could more clearly benefit from improvement. By ‘traditional development processes’ we refer to the most common software development practises (including methodologies, tools and languages) that are commonly used by industry nowadays, especially when we refer to non-trivial management information systems.

Secondly, this thesis aims to study the mentioned development practises in the light of the cutting-edge requirements of software engineering (i.e., interoperability, distributability, scalability, reliability, etc.), in order to evaluate how well they can cope with the new technical challenges that must be met nowadays. While attention has been generally paid to the resulting products, here we turn to the actual process and address the question of whether good software could possibly be produced in better, faster, easier ways. For years, all efforts were on defining the most advisable methodologies to follow, the most effective approaches to requirements elicitation, the most convenient cycles of system analysis, design, implementation and testing stages. Instead, we want to focus on the paradigms and tools on which those methodologies and techniques have been unquestionably relying, and determine whether and why there can be a better choice for them under the previously-mentioned circumstances.

It is also the intention of this thesis to propose, then, a new perspective to software development, based on the adoption of an alternative paradigm. Similarities and differences with the new model will be explained in depth. Properties and incentives of the new approach will be exposed, as well as the number of advantages that are derived from it. These will be showed on the basis of a real case study that will serve as story line for this dissertation.

Last but not least, we intend to formalise our proposal and state, as the final contribution of this thesis, a new functional software development methodology. Global benefits with respect to imperative/object-oriented-based development cycles will be detailed, particularly stressing the general tasks where a functional strategy represents a substantial improvement for the development of applications on complex business domains.

1.3 Structure and contents

This document has been divided in chapters, each of them with a specific purpose which clearly reflects in its contents.

After this introduction, the state of the art is explained in Chapter 2. A brief review of the most popular software development methodologies has been written, as well as some considerations about their strengths and weaknesses, together with open problems, unsolved issues and remaining challenges mostly related to business modelling. A compare-and-contrast section on the imperative and the functional programming paradigms follows, to thoroughly examine advantages and inconveniences of each perspective. Finally, some remarks are made regarding the specific process of software validation and testing, one of the aspects of software engineering where there is still great room for improvement.

The third chapter is devoted to introducing the system that has played the role of case study for this thesis. It is a real-world application, currently in production for a few years already, that was completely developed using the methodology that we generalise and formalise here as our functional software development methodology proposal. The business area that this software addresses is risk management in the insurance field, whose main concepts are introduced as well, in order to provide the reader enough knowledge to be able to fully understand the use cases and examples extracted from the case study that will illustrate the following chapters.

The rest of this dissertation has been structured according to the main stages present in any software development process, which have been grouped in three big sets of activities. First, the sequence of interactions that involves requirements extraction, continues with system analysis, and leads to the operative design. Secondly, the core tasks concerning the actual creation, building, and implementation of the system. Finally, the very important processes of software testing, a key aspect in the pursuit of product quality and an essential element in improving the maintenance stage, probably the longest in the lifespan extension of any application or system. Each and every of these three central chapters is closed by a section which specifically refers to our use case, to show how the arguments previously made apply in reality. The intention of this layout is to both conveniently translate theory into practise, achieving a better understanding of the related concepts, and to present some empirical proof of the points made.

To sum up, Chapter 7 gathers together all the considerations about the software development process from the previous pages, and presents, in a nutshell, the specifics of the functional software development methodology that we propose. All the comments about the different stages, as well as the strategies and recommendations that form our contribution, are explained from a higher level perspective, describing the final outline of this methodology.

The thesis closes with a final chapter, a complete evaluation of the work that has been done and the contribution it represents to the current status of the field of software development of complex information systems, with a special insight into the functional programming and the software validation worlds.

Some thought has also been devoted to open research lines and future work to be done, which are included here as well.

2
State of the art

This chapter presents some general concepts that need to be clarified in order to understand the objectives and relevance of this thesis. Background information about software engineering, its methods and practises, as well as an analysis of its current situation is brought into the picture, to provide the reader with the needed context. The challenges that remain open in the field are examined and related with the purpose and goals of this work, with special attention to two main aspects: the approach to design and implementation, and the carrying out of software validation.

2.1 Software engineering

Software engineering is a discipline devoted, as all engineering, to solve problems from our daily lives. Specifically, software engineering is concerned with all aspects of software production. Even though software construction was not considered an engineering process for a long time, it has proved to possess all the properties, risks, and requirements to be expected from an engineering activity [7].

At a sustained pace, social demand for software solutions applied to an increasing number of fields is growing day after day [8]. To create those software applications, the problems they are expected to solve need to be carefully and exhaustively examined by experienced professionals, in order to gather both the fundamental essence and the important details of the matter. Then, all alternatives have to be considered, evaluated, and weighted, discarding the

least convenient and choosing the best technical options for each case. Last but not least, this technical proposal shall take shape in the form of a specific piece of software, and its functionality and suitability have to be ensured before it can actually go into real operation.

Software engineering professionals are required, thus, to use their particular technical and scientific knowledge and skills to analyse problems, divide them into as many confrontable subproblems as appropriate, creatively invent singular and precise solutions for each of them, and then successfully combine all those partial solutions into the final and complete one. Besides, the engineering process of finding and creating a solution for a given challenge has to meet certain quality criteria, which will ensure that the engineering response to the problem reflects the best possible effort.

To face their engineering duty, software professionals use and apply a broad set of methods and techniques. We use the term *method* or *technique* to describe a formal procedure for obtaining certain results, which is usually carried out with the help of specific tools. While the combination of auxiliary tools and instruments may be part of the procedure, it is actually the combination of activities that defines the process, and the tools and instruments are just automated systems to improve the accomplishment of those tasks by helping to make them more productive, efficient, or accurate, leading to a better quality result.

The main activities involved in the creation and management of a software product are [9, 10]:

- *Software specification*
- *Software development*
- *Software validation*
- *Software evolution*

The construction of any system or application starts with its conceptualisation and specification. The global objective of the software has to be determined, as well as its properties and characteristics (functional and non-functional, i.e., including performance demands, usability constraints. . .), and the functionalities or services it will need to present or provide. This process of requirements analysis and definition is critical to the success of a development project [11].

Once the goal system has been pictured, it is time to start its actual development. Of course, this is the core activity in the software development project [12, 13], and it can generally be divided into two broad stages: design of the proposed system (including its architecture, environment, structure), and implementation in a given platform and programming language.

While it can be seen as part of the application development or as a separate task, the third of the former activities intends to provide some feedback about

the product quality and its degree of commitment with the project aim. Software testing can be performed at a number of different levels [14, 15], from the smallest application components (i.e., unit testing) or the different application subsystems (i.e., integration testing) to the proper operation of the entire system as a whole (i.e., system testing), and the fulfilment of end user expectations (i.e., acceptance testing). This validation process assesses whether the system is suitable according to its purpose.

Eventually, the system will be ready to start working on a production environment, but the life of a software product does not end when it is handed over to the customer [16]. After the system delivery, continuous maintenance and support is very likely to be required, including not only solving the minor problems that may still appear, but also the enhancement and evolution of the application, in case new needs arise.

Any combination of all or some of these activities, organised so that they constitute a guide towards the production of operational software, is what we call *software development process* [17]: a structured approach to software development whose aim is to facilitate the production of high-quality software in a cost-effective way.

2.1.1 Software development methodologies

In contrast to a process, which usually states the set of actions involved in the development of an activity, a *methodology* is a more specific series of steps to be followed when an intended result is pursued, an ordered sequence of tasks which sometimes specifies also tools and/or techniques. Traditionally, when it comes to software development, the methodologies to build software products have also received the name of *life cycles* [18].

Software development life cycles are important because they bring a consistent structure to the software building activity, making the construction of an application a repeatable and knowledgeable process, thus constraining risk and making the whole process manageable and dependable. An immediate benefit of following a structured methodology when designing and building software is the possibility to graphically explain to the customers, who are usually rather unaware of software construction complexities, the steps on the way to develop their system, so that they can become more familiar with the software development process and all the stages involved.

The first software development life cycle was formalised by Winston W. Royce in 1970 [19], as a systematic and purely sequential series of stages which included, in this order (cf. Fig. 2.1):

- system requirements extraction
- software requirements extraction

- analysis
- program design
- coding
- testing
- operation and maintenance

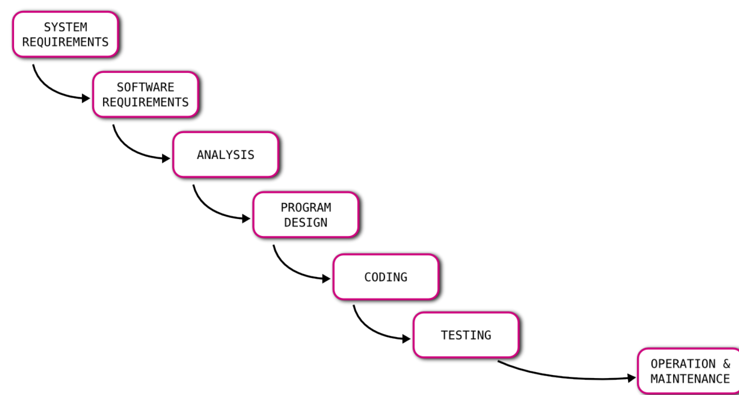


FIGURE 2.1. Waterfall software life cycle

Many of the software development approaches that followed since the formulation of this **Waterfall model** can be considered just as modifications on the main schema outlined by this first life cycle, adding room for extra activities, feedback loops, and iterations. Nevertheless, as the oldest development model in software engineering, it is still considered the "classical software development".

One of the first variants introduced on the Waterfall model was the addition of prototyping [20] (cf. Fig. 2.2).

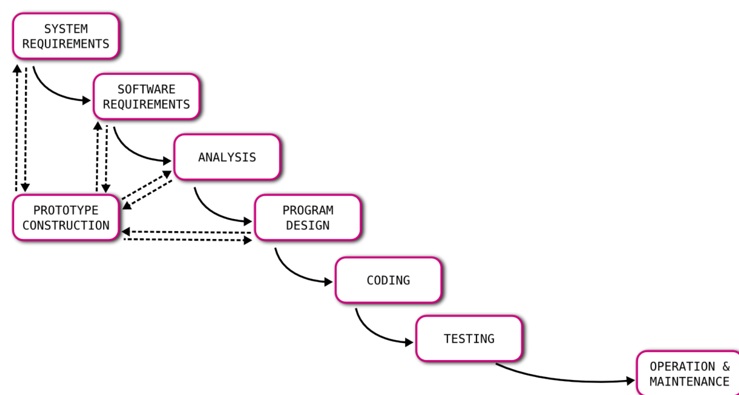


FIGURE 2.2. Waterfall with prototyping software life cycle

The major drawback of the original methodology was soon enough argued (and stated by Royce himself), and lies in the fact that hardly ever the real evolution of a software project is as clean and sequential as the model formulated. A *prototype*, as a partially but quickly developed product, allows the customer to examine different aspects of the future system still at a primitive stage, and hence to decide about its suitability. This early validation is very important to obtain some decisive feedback about the product which is being built and the correct interpretation of the user needs and system requirements. Incorporating prototypes was an effort to achieve a closer, and thus more reliable approximation to the reality of projects. However, the model was still too rigid, so further proposals continued to appear, as the need for a sounded methodology was growing clear.

The possibility of introducing modifications in a project whenever a failure or misconception was detected, no matter whether it was on the requisites specification, the system design, or the program implementation, resulted in the appearance of the **V model** [20] (cf. Fig. 2.3).

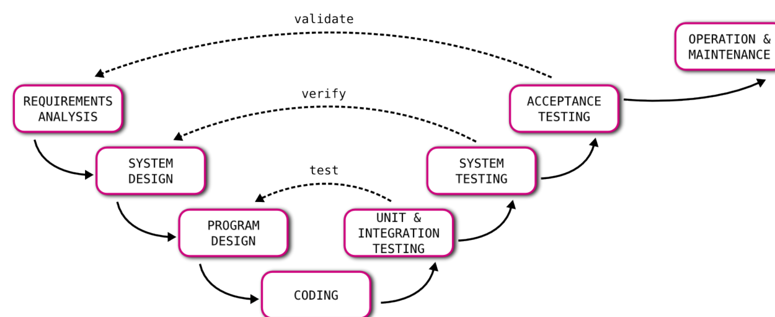


FIGURE 2.3. V software life cycle

This new refinement of the Waterfall methodology tried to illustrate how the different activities of the software development process were related to each other, specially linking the different *creative* tasks (requirements elicitation, system design, program implementation) with the corresponding *testing* stages. These relationships were made explicit by this new model, so that on appearance of a problem, detected at certain level of the 'upward branch' of the process, the corresponding step on the 'downward side' is the stage that needs to be re-executed in order to solve the problem.

Even though the innovative idea of bearing in mind where an error might have originated was interesting, the perspective of just repeating that stage to amend it was not the most efficient one, and how the project should go through the rest of the life cycle again was not formally specified either. Still, this proposal had great influence on subsequent methodologies which explicitly exploited the notion of iteration that was already underlying the V model.

The **Prototyping model** tried to improve the Waterfall approach after the prototype strategy was introduced [21]. The intention was to take advantage of the prototype construction process that helped through the first phases, by making it evolve into the final product instead of just using it for early evaluation and then discarding it (cf. Fig. 2.4). Apart from making the most of all the efforts put in the project from its early stages, working on the prototype to turn it into the final working application also helps to reduce the project failure risk and uncertainty. However, other kinds of risks appear which constitute the main criticisms of this methodology: insufficient analysis, incomplete specifications, poorly engineered final result.

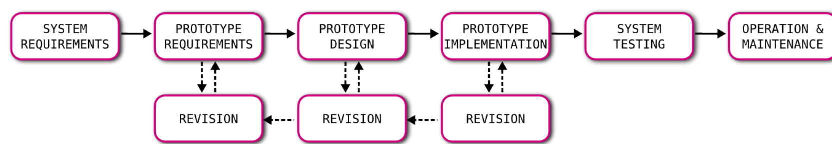


FIGURE 2.4. Prototyping software life cycle

As phased development models were popularised, a set of methodologies based on iterations and increments instead of linear sequences of activities appeared, as part of the continual effort to improve the time to market of software products. It was already clear by then that the software industry was a promising business activity, so the urge was great to find the best operational model for software construction. The greatest novelty of these new models was the coexistence of two different versions of the software system in parallel. On the one hand, there was the application under development; on the other hand, there was the previous operational version of the software, already put to a production environment. To handle this simultaneous operation, the methodologies need to produce a working version of the goal system as soon as possible, and then continue working on new releases, adding functionalities and improvements, and also correcting errors. One of the most well-known of these is the **Incremental development model** [22].

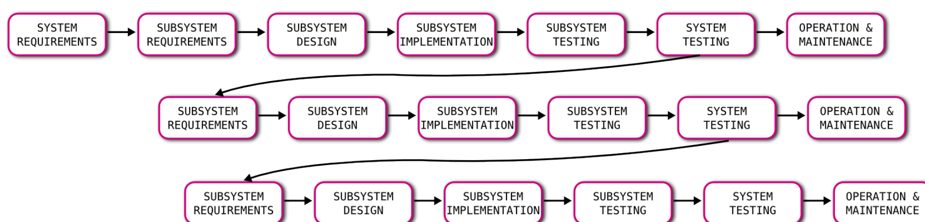


FIGURE 2.5. Incremental software life cycle

This software life cycle suggests the grouping of system requisites by similarities, in order to architecturally divide the application into as many subsystems

as groups of related requirements. Beginning with a small operational system with one or a few working subsystems, this development model continues to make the system grow and evolve up to the final version, by adding new functionalities (i.e., subsystems) in each new release (cf. Fig. 2.5). In some sense, the Incremental model can be seen as the superposition of several Waterfall developments.

Very similar to the Incremental model, the **Iterative software development model** [22] differs from the previous in the way the final software product is approached: instead of adding more and more functionality with each new release, a rudimentary, not optimised version of the full system is delivered already as first version. Then subsystems, their services and functionalities, are improved with each new release, so that the whole set is enhanced, step by step, until the product reaches the standard of quality of the project and the terms agreed with the customer.

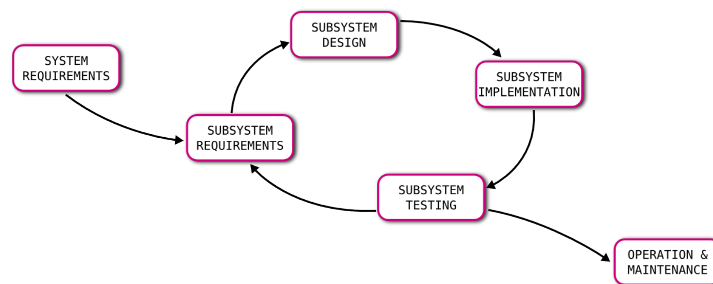


FIGURE 2.6. Iterative software life cycle

In practise, many organisations have used over the years a mixture of the Incremental and the Iterative software development methodologies.

Sixteen years after the Waterfall model was described by Royce, in 1986 Barry Boehm introduced a revolutionary aspect into the software development process [23]: the **Spiral model** (cf. Fig. 2.7). Using an iterative methodology as basis, he argued that the evaluation of risks should be explicitly considered among the stages of a software project, so that not only different prototype alternatives could be evaluated, for instance, but also other potential dangers or threats to the successful completion of the project itself. In other words, project management had finally made its appearance in the world of software development.

By the beginning of the 1990s, it was clear that iterative processes produced better results than their linear predecessors, but there was a need to make software development a more dynamic process and to avoid at least some of the overload caused by documentation duties and other administrative tasks within most project development processes.

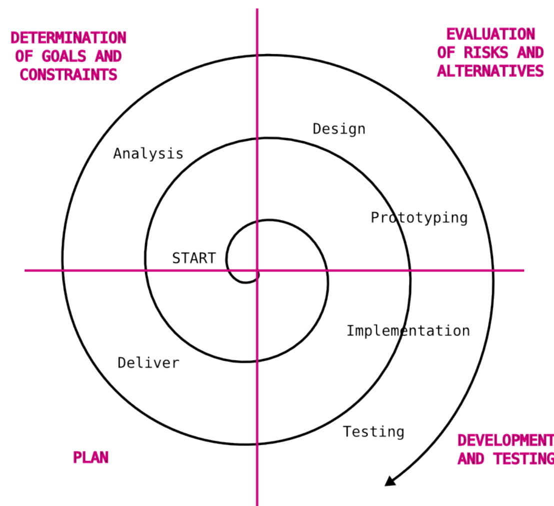


FIGURE 2.7. Spiral software life cycle

In 1991, James Martin formulated the concept of **Rapid Application Development** (RAD) [24] (cf. Fig. 2.8), as a new philosophy which main focus was to pursue shorter development cycles (and thus, higher development speed). Still following an iterative approach, the RAD methodology introduced explicit scheduling of communication and planning activities as part of the life cycle, since it encourages the existence of different development groups working in parallel in different parts of the system. Even though Martin's proposal was not free from criticisms, especially related to both application and development teams size, it would inspire the next generation of software life cycles.

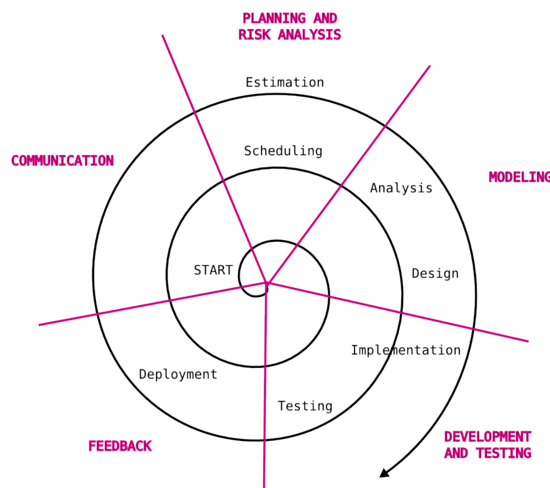


FIGURE 2.8. RAD software life cycle

It was in the late 90s when the most recent movement on software engineering and development finally took off. With the unquestionable and unstoppable democratisation of information technologies, the number of software development projects had increased enormously over the last decade, and both developers and managers found themselves in need of more flexible ways of organising and working. Following the steps of RAD, they chose to put more value on individuals, their intercommunication capabilities and technical skills, instead of devoting so much effort to management and documentation. Attention was shifted from processes to people by a whole set of new methodologies named '**Agile methods**' [25]. The common philosophy behind this innovative trend was the genuine intention of investing all the time and resources in actually producing good software, instead of filling in tons of reports and documentation that in the end nobody was reading. They replaced the paperwork by more active collaboration with the client and within the working teams, and suggested to focus on responding to changes, rather than on creating and following a plan from beginning to end, trusting the developers to organise themselves and encouraging face to face communication over documentation interchange.

The most popular of these new agile approaches is **Extreme Programming** (XP) [26], up to the point that both terms, *agile process* and *extreme programming*, have become almost synonyms in practise. XP (cf. Fig. 2.9) claims to be based on four properties: communication (all project members, including users, need to share the same view of the system), feedback (referring to both people interaction and system output) for testing, simplicity (simple designs and code are preferred), and courage (the previous properties rely on the participant's ability to comply with them). Paradoxically, the strengths that XP claims for itself are the more controversial aspects of the process, such as flexibility/instability of requirements or potential personal conflicts.

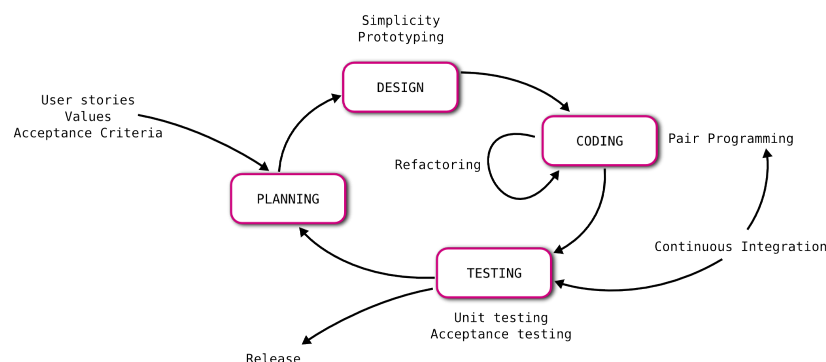


FIGURE 2.9. Extreme programming software life cycle

As the agile processes had pointed out, permissiveness of change turned out to be the primary driver for adaptability and evolution of software development methodologies. In 1999, Ivar Jacobson, Grady Booch, and James

Rumbaugh described a life cycle in which they tried to draw the best features of conventional software process models and characterise them to implement the best principles of agile development: the **Unified Software Development Process**, or Unified Process for short [27] (cf. Fig. 2.10).

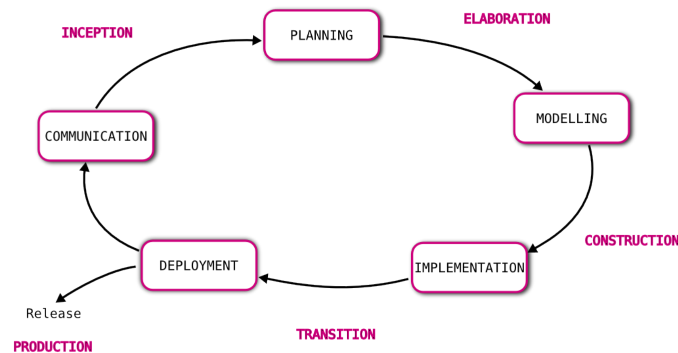


FIGURE 2.10. Unified Process software life cycle

This *hybrid* model is the first to describe the software development process as a combination of three different views or perspectives: a dynamic perspective that shows the phases of the model over time (inception, elaboration, construction, transition, and production), a static perspective that shows the properties of those process activities, and a *praxis* perspective that suggests good practises to be applied during the process.

Apart from the most remarkable software development methodologies that have been mentioned here, other minor and specialised methodologies and life cycles have also been proposed for specific situations or critical projects [28–30]. Some of them are, for instance, Aspect-Oriented Software Development (AOD), Adaptive Software Development (ASD), or Feature Driven Development (FDD). All of them, nonetheless, are based on or inspired by one or several of the software processes that have just been discussed here, including or omitting only specific tasks or activities to better suit their particular needs and goals.

2.1.2 Software quality approaches

Good software is defined by a set of properties that includes maintainability, dependability (involving not only reliability, but also security, and safety), efficiency, and usability [31]. Common adoption of software development methodologies was a huge step towards a greater quality of software products, but no process model includes explicit and specific recommendations to increase software understandability, verifiability, repairability, robustness, performance, re-usability, portability, or interoperability, among others; and those which claim to do so, just include some generic advice.

A number of tools, commonly referred to as CASE tools (for *Computer Aided Software Engineering* tools) provide automated support for requirements analysis, system modelling, debugging, and testing activities [32, 33]. Their use can make developer's and engineer's lives easier, but does not bring quality as a direct result. The quality of the product, especially when developed using certain methodology, is closely related to the quality of the development process itself. In the last decades, the belief has spread that more efficient software processes result in products with better quality. As a consequence, many software engineering companies have turned to software process improvement as a way of enhancing the quality of their software. This has been the origin of enhancement movements such as CMM℠/CMMI© or standardisation rules such as ISO 9000/9001:2008.

The **Capability Maturity Model** (CMM℠) [34, 35] is a set of system and software engineering capabilities that should be present in an organisation and its software development processes in order to be able to accomplish effective process improvement. Originally developed by the Software Engineering Institute (SEI) at Carnegie-Mellon University in 1989, CMM℠ has now been superseded by the *Capability Maturity Model Integration* (CMMI©) [36–38] for software development; it is claimed, based on empirical evidence, to help improving predictability, effectiveness, and control of organisations' software processes.

CMMI© is both a recommendation of good practises for software construction and maintenance, and a kind of process meta-model that classifies a development process in one of five evolutionary levels (Initial, Managed, Defined, Quantitatively managed, and Optimised), depending on the level of management and control an organisation has over its own processes in terms of documentation, repeatability, consistency of results, standardisation, use of metrics, improvement policies, etc. CMMI© has been criticised for the considerable amount of bureaucracy that it introduces. In addition, this meta-model does not really help define the structure of an effective software development organisation. The behaviours and best practises recommended are just properties that have been detected on successful projects, but being CMMI© compliant does not necessarily guarantee the success of a specific project.

The **ISO 9000** series is a set of standards for quality management systems which are maintained by the International Organization for Standardization (ISO) [39]. The ISO 9000 family includes rules for fundamentals and vocabulary (ISO 9000:2008), for requirements (ISO 9001:2008), and for performance improvement (ISO 9004:2008), among others. In particular, the ISO 9001:2008 standard, intended for any organisation which designs, develops, manufactures, installs, and/or services any product or provides any form of service, can also be applied to software development companies. This standard demands:

- A set of procedures that cover all key processes in the business.
- Monitoring processes to ensure the organisation procedures are effective.
- Adequate records being kept of procedures and their performance.
- A checking organisation that processes output for defects and performs corrective actions when necessary.
- Effort on continuous improvement.

This set of requirements needs to be fulfilled if a company is to achieve customers satisfaction through consistent products and services which meet their expectations. And even though they are perfectly applicable to software development, they are also so generic that they do not add any useful information. Besides, ISO 9001:2008 may be a generic standard to improve overall quality of products, systems, or services a company provides, but it certainly does not take into account any of the software development peculiarities. In the same way CMMI© compliance does not guarantee the success of a software project, certification to the ISO 9001 standard does not guarantee any quality of end products and services; rather, it just certifies that formalised business processes are being applied.

The main difference between the two systems lies in their respective purposes: ISO 9001 specifies a minimal acceptable quality level for software processes, while the CMM establishes a framework for continuous process improvement and is more explicit than the ISO standard in defining the means to be employed to that end. Improving a process means being able to first fully understand it and to later on change it in order to better pursue a specific goal. Most of the literature on software process improvement is focused on perfecting the development processes; however, the actual quality improvement should not simply mean adopting a particular methodology or tool, or using some specific model, but to globally make the software engineering process more trustable.

2.1.3 Open challenges

So it should be clear by now that there are a number of challenges in software engineering that remain open to date [40, 41]. Some of them are related to software itself, some refer to the software development process.

When it comes to software, there is a series of characteristics that have turned from features to essential properties along the years. They can be grouped into three main branches [1]:

- *Heterogeneity*.
  Systems are increasingly required to distributedly interoperate across heterogeneous networks, interacting with new and old legacy systems,

written in different languages, running in different environments and operating systems. The challenge is to develop techniques to build dependable software, flexible enough to face these demands.

- *Delivery*.
  With traditional software engineering techniques, software quality was revealed as a very time-consuming property to acquire. On the other hand, businesses are responsive and change quickly, and so must their supporting systems. The challenge is to achieve short delivery times for large and complex systems without compromising quality.

- *Trust*.
  Software is present in all aspects of our lives, some more critical than others. Still, it is very important, not only that systems can be trusted when it is really crucial, but also that users perceive them as generally reliable, so that they can be fully accepted as part of their daily lives. Of course, this can only be achieved through development of quality products.

Undoubtedly, these open challenges are interrelated, which means that efforts in any of these three directions will reflect on all of them. It also means that the wiser approach is likely to be a wider perspective and approximation to all of them, the final objective being to provide engineers, managers, and developers with the best possible set of choices to achieve their goals (i.e., successfully develop their software projects) with the most efficient usage of time and resources. Or even more importantly, avoiding that certain projects may not be undertaken due to the unawareness about all the tools and methods that may help to address and accomplish them.

2.2 Imperative paradigm vs. Declarative paradigm

In contrast to a software development methodology, which is a style of solving specific software engineering problems, a *programming paradigm* is a fundamental style of computer programming [42]. Paradigms differ from each other in the concepts and abstractions they use to represent the elements of a program, and the steps that conform computation.

The *imperative programming* paradigm describes computation in terms of statements that change the state of the program. Imperative programming, thus, is based on sequences of commands to be performed by a computer. The term 'imperative' is used in contrast to the '*declarative*' *paradigm*, which expresses what needs to be done, rather than how to do it in terms of series of actions to be taken [43]. Functional programming is one example of a declarative approach.

The use of one paradigm or another not only has a great impact on the election of the programming language to support the development of a system or appli-

cation, but also on what the actual code looks like and how it is implemented. Traditionally, software development life cycles have implicitly relied on imperative programming practises, and CASE tools are fundamentally oriented to them. However, from an objective point of view, a declarative approximation is much more suitable to solve complex problems, like the ones we are forced to face more and more frequently. Challenges in software engineering have moved from algorithm design to business logic and constraints implementation. With a paradigm that does not constrict engineers to think about *how* they have to specifically solve a problem, but instead raises the abstraction level of the development and allows them to concentrate on *what* needs to be accomplished, the whole reasoning process stays at a level which is much closer to human problem perception and argumentation, and also to problem specification and description. In addition, a system which is not only created and designed at this abstraction degree, but is actually implemented using a high-level declarative programming language, will enable validation activities to present the same properties, thus allowing the easier performance of powerful verification tasks.

Still, there are imperative artifacts that can really improve the results of the analysis and design activities in a software development project [44–46]. In particular, concepts from the object-oriented programming paradigm (OO), the most popular imperative approach nowadays, are indeed useful. OO uses the notion of “object” to model concepts from the real world. Interactions among those objects (in the shape of message interchanges) are intended to represent dynamism and business activities. However, while objects are generally a very good abstraction of business elements and data, it is not that often the case that modern business cases and logic can be easily depicted as just a set of messages between those objects. Rather than being *active* elements, in many situations engineers will find themselves creating artificial solutions in order to design communication protocols for intuitively *passive* objects, in order to carry out system functionalities and services.

On the other hand, declarative approaches such as functional programming explicitly try to avoid state and mutable data, which is also an unnatural and unwanted circumstance in most business scenarios. While its proximity to mathematical foundations, the use of pattern-matching, high-order functions, and recursion, can be of great help to easily implement logic and operations in an intuitive and descriptive-like manner, there is a lack of resources when it comes to data modelling and management.

Ideally, we would like to be able to take advantage of some of the OO benefits, but without renouncing the functional properties of abstraction and high-level reasoning.

2.3 Software testing: verification and validation

We have already mentioned on several occasions the fundamental need for testing efforts as part of the software product development process [47–49]. Software testing is an essential activity which tries to answer two different questions: "*is this the right product?*" (Validation) and "*is this product right?*" (Verification).

Validation, thus, is intended to decide whether or not a product design satisfies the intended requirements and meets the customers' needs. Generally, to determine this, a system undergoes different sets of external manually-performed operational tests, either by potential users or by testers. Verification, on the other hand, is meant to inspect whether or not a product has actually been built according to its design and specifications.

There are some tools that can help performing verification duties, such as code and data analysers, structure and sequence checkers, program monitoring tools, test case generators... But in order to really effectively and efficiently detect and solve problems within a software project, it is the testing techniques and tactics they are used for which are important.

2.3.1 Unsuccessfully addressed problems

There is a set of software engineering principles that are widely accepted among the software development community [50–53]. These principles include rigour and formality, separation of concerns, modularity, abstraction, anticipation of change, generality and even incrementality. Software life cycles take into account most of them, and they also include the testing activities that should provide some confidence on how close to those principles has the project been performing.

Yet, no software development process states explicitly which techniques or methodologies are best followed to actually perform meaningful testing, so in practise the related activities are faced in a much more *ad hoc* and manual manner than would be advisable. This, of course, affects not only the time and resources they require, but most importantly, their suitability and effectiveness. There are no significant metrics that we can apply to decide how accurate a verification or validation stage is or has been, and neither are there well-grounded, sound criteria to help us decide when to stop performing tests.

In conclusion, there is a lack of formality when it comes to software testing that needs to be addressed. Development teams of all sizes could hugely benefit from having guidelines or defined procedures to follow when testing their products.

3

Case study

The development of this thesis has been intimately related to the conception, creation, and evolution of an information system used as a case study. Therefore, before proceeding any further, the business case, features, and properties of this application are explained in this chapter.

First of all, some basic notions about the field of risk management and some discussion about the specific issues of the insurance domain take place. Then, previous existing and alternative software applications to support this activity are discussed, focusing on their particular properties and weaknesses, which eventually lead to the birth of our case study project. Finally, we present ARMISTICE, describe the system, the particular problems it is designed to solve, and the improvements that it is conceived to make.

3.1 Risk management

Enterprise Risk Management (ERM) is a field of enormous importance due to its increasing complexity and undeniable economic value. More and more companies are paying attention to it, given there is a significant economic reward for attending to the various aspects of risk [54]. Besides the recognition of the value in ERM, there is also an awareness of the need to apply more complete, integrated approaches. The discipline has witnessed a shift in the way organisations manage the many uncertainties that stand in the way of achieving their strategic, operational, and financial objectives. “Band-aid” approaches to risk management – with each risk considered in isolation and

only when it occurs – have been replaced with more holistic methods, looking at risks as they are integrated and interrelated across the entire organisation, and managing risk response strategies well before they become acute [55].

All enterprises are different in some sense: their business models differ, the types of products and services life-cycles are context-driven, organisational charts are diverse, their motivations for overall business style are not the same. However, most of them have as their bottom line the same pursuit of economic success, so their *objects of interest* may not be the same, but their ultimate aims are. Any company interested in dealing with risk needs to define a set of considerations, of guidelines and intentions, that comprises the company's policy on ERM. To face hazard threats, there are different risk management strategies that can be applied, different philosophies to follow, different attitudes to adopt (namely, avoidance, prevention, assumption, or transference [56]). And of course, the best risk management policy always depends on the specific company, its business area, its particular situation on the market, lifetime, size, etc. Nevertheless, it is often a wise choice not to apply only one of the previous strategies, but to build a customised risk management policy choosing for each risk the best attitude in each particular case.

3.2 Risk Management Information Systems

So ERM has become a matter of real importance for the enterprise today, an aspect of decision making for every CEO. This is a direct consequence of the new business concept it introduces: economic prosperity is not just about making money, it is also about avoiding losing money. And this means, most of all, the intimate overseeing of all company resources (whether they be human, material, or ideological) and the close monitoring of all its activities, also from a preventive point of view. In the business world, the main objective has always been maximising the success of the enterprise objectives through carefully planned strategies of action. With ERM, one must be take into account that it is equally important to ensure we protect business activities from failure due to external or indirect causes that may go unnoticed, at least to the non-expert eye. The incorporation of risk management then enhances the overall economic objective by expressing risk not just as a threat, but also as an opportunity for economic profit [55].

Risk management theories and procedures are formulated and re-formulated every day, presented and discussed in major conferences and meetings [57–60] at an international level. Risk management associations and organisations [61–63], for the exchange of ideas and experiences, draw membership in the thousands. More generic decision-support forums show their interest about the subject [64], too. But even though we now assign risk management the importance it deserves, and despite all efforts that are being made to face its potential threats, it is obvious that we have not employed all our potentially useful tools against it. In a society where information technologies are more

and more present in the daily life of business and economy, ERM seems to be one field where software engineering, unfortunately, has not made an impact yet. Personal computers are part of the daily routine of risk management departments everywhere, but user-level usage of computing is as far as it gets. This restrains a potential benefit, not only from applying automation and computing power to repeatable processes, but from using software engineering techniques to create new software tools, to solve problems on the whole that are not being adequately addressed by existing systems. Instead of using computers just as an auxiliary tool, the goal should be to use them to take over as much of the risk management tasks and processes as possible.

3.2.1 Commercial software for risk management

The majority of the software systems for risk management, usually referred to as Risk Management Information Systems (RMIS), available in the market are conceived to be used by insurance agents, brokers, and carriers (cf. Table 3.1). In other words, their focus is on the intermediate actors that study and recommend the best insurance options to their clients, among their insurance portfolio, and are not generally designed from the insurance customer point of view. As a consequence, most RMIS do not reflect the needs and use cases of insurance holders, but those of the intermediate agents that stand as negotiators between insurance companies and their potential clients.

For small companies and private or individual interests, hiring insurance brokers is generally a good option. Their knowledge about the insurance business and the different options in the market, together with their risk-related experience, will be an advantage worth benefiting from in most cases. However, when we talk about larger organisations, with international cross-borders activities, with critical and/or strategic products or services that need to be privately managed, with an extensive amount of objects of interest, outsourcing risk management does no longer seem a reasonable alternative. These kinds of insurance clients typically have their own risk management department, which of course have some slightly but fundamentally different software needs.

An information system meant to be useful in this scenario should provide mechanisms to define the specific resources of the company, and manage them from the point they appear or are acquired to the moment they are disposed or are not of interest any more. The organisation also needs means of handling all the information about the insurance policies it may engage, such as coverage details, terms of protection and compensation, etc. And since the objects of interest are not likely to be static entities, their changes probably affect the insurance policies, whose specifications may also be modified as a consequence. Hence, policy supplements and renewals are required to be automatically managed, too. Last but not least, the most important feature of an RMIS from the insurance client perspective is the administration,

Product name	Vendor	Platform	Orientation	Features
Insure3 Policy, Insure3 Billing, Insure3 Claims	Castek Software Inc.	MS Windows	Property and casualty insurance.	Complete risk management is offered to insurance carriers, but requires the use of the three applications.
My Insurance Center	Cover-All Technologies Inc.	MS Windows	Adaptable by vendor.	Facilitates management of offered insurance options to brokers.
Web Based Insurance Management System	Radixweb	Web	Adaptable by vendor.	Manages policy registration, claim processing, and documentation.
DaeSegur	Daeda Solutions	Web	Car insurance.	Manages policies, receipts, accidents, and agenda for insurance agents.
SoLife, SoLiam	Business Solutions Builders (BSB)	Multi-platform	Life and financial insurance.	Offers a great variety of configuration options and functionalities.
Is-CliSeg	Infoseg S.A.	MS Windows	Designed for small-scale insurance companies and insurance brokers clients; inventory management and feedback reports.	Designed for small-scale insurance companies and brokers, offers inventory management, policies, payments, invoices, accidents, and claims support, as well as simple reports.
Stars	Marsh Inc.	MS Windows, Web	Customisable by vendor to fit client's field and role.	Large-scale oriented, manages risk data and claims according to user roles, supports multi-language and multi-currency, and offers intensive reports.

TABLE 3.1. Commercial Risk Management Information Systems (2008).

control, and supervision of accidents which affect their objects of interest, as well as the corresponding claims and evolution of these from accident report to damage reparation and loss indemnification (including payment orders, invoices, and handling of all associated documents). As one can imagine, all these tasks and activities involve large amounts of data; thus, additional data analysis tools and report generation features are also of maximum relevance for managers, who need to be able to study the performance and efficiency of the current ERM strategy of the company.

3.2.2 Prospectives

Given that ERM recognises that organisations face a greater variety, and an increasing number and interaction of risks [65], it is surprising that the software applications which claim to help to manage the risks are not designed to cope with these enormous differences, even though their commonalities are the really important point.

From our previous analysis of the existing commercial risk management tools (cf. Table 3.1), we conclude that there is a need for a new sort of RMIS to fill a significant void. A different system, designed with one important thought in mind: to be powerful enough to model all the complexity and diversity of the risk management process, but also flexible enough to be *self-adaptive to any company and therefore any type of risk*, regardless of its particular business domain. By 'self-adaptive' we mean that the system itself should have the built-in properties and features to be used, right out of the box, by any insurance client, without requiring any further tuning or human intervention (in particular, developer's intervention). In other words, a system where customisation to a specific business activity could be easily performed by the users, as part of their regular work.

This innovative RMIS needs be designed to be a tool for different user profiles, according to the various degrees of expertise that would be present among the staff of a risk management department. For the expert user, it should be of help to spell out the company's specific objects of interest, and their relevant properties from an ERM point of view. This user profile is likely to be responsible for defining the insurance policies contracted to protect those resources from the consequences of potentially harmful events, whichever these might be, for each particular case. For the non-expert user, who probably has to deal with accident reports and tracking having little or no advanced knowledge regarding coverage and warranties, the system should be able to provide *active support in the decision-making process*, retrieving and isolating only the most relevant information in each case, according to the contextual data provided, and thus, providing extremely valuable support for final decisions.

Of course, this software product should maintain all the interesting properties that we have already identified in the systems we have analysed: multi-user

capabilities, multi-platform usage, multi-database compatibility, multi-language configuration, multi-currency support, powerful report functionalities... In addition, it should also face all the issues that we outlined in Section 2.1.3 (page 20): distributability and interoperability, timeliness and quality, robustness and fault-tolerance.

These, as we will see in the next section and the following chapters, are the challenges to be faced by our case study, ARMISTICE.

3.3 The ARMISTICE project

ARMISTICE [66–68] (*Advanced Risk Management Information System: Tracking Insurances, Claims, and Exposures*) is an efficient and robust risk management information system (RMIS) developed using the advanced software engineering techniques and methodologies proposed in this dissertation, which conferred this software unusual and powerful flexibility, as well as high reliability.

The system is a three-tier client/server vertical application which is able to, among other things:

- Model and manage all kind of organisation resources.
- Model and manage all kind of potential risks.
- Model and manage contracted insurance policies to a fine-grain level of detail (by means of its own high-level language based on formulae and constraints).
- Manage the claims for accidents involving resources all over the world.
- Select the most suitable warranty to cover resources damaged in an accident (help decision support expert system).
- Manage other accident-related tasks (payments, invoices, repairs...).

As it is easy to see, ARMISTICE's application domain is quite complex. Many entities, with complex relations among them, are identified in the analysis of the domain. Thanks to the level of abstraction reached at the definition of the system, which is that of meta-information, great versatility can be achieved. When talking about meta-information, we mean that the system deals with information about information, that is, it is designed not only to be able to manage some specific concepts, tied to more or less specific cases and specific business scenarios, but with the ability to specify the very nature of those concepts, cases, and scenarios in the first place. Besides, the high abstraction level reached when designing the system is maintained through the development stage thanks to the use of the functional programming paradigm, easing and speeding up the implementation process, and also enabling some advanced validation strategies to be applied during testing activities.

This makes it possible for ARMISTICE to be applied to diverse business fields, regardless of their nature, and makes it a perfect case study to show empirical evidence about the suitability of the methodology applied to develop this software in order to approach other complex problems, conceive complete solutions, and successfully perform them.

3.3.1 Data collection and analysis

All the information and data about the ARMISTICE project that is included in this dissertation has been gathered by the development team during the lifetime of the application, from its early conception stages till the present day. The analysis of the data was conducted in a exhaustive way, sometimes trying to extract significant hypotheses, sometimes looking for arguments to support our own theories. Whenever a conjecture was formed, further data was inspected (or additional data was acquired) in order to validate whether or not there was any evidence in favour or against it.

Different techniques have been used for data collection, that can be grouped in two sets:

Scheduled data collection Some information was obtained as a result of a planned activity or effort, such as interviews with the users or development team meetings. Two kinds of data profiles can be identified, the first being unstructured data (i.e., informal notes), and the second being structured data (i.e., design documents in the shape of UML diagrams).

On-demand data collection Apart from the information that was gathered as a result of scheduled tasks or events, an important amount of data has been collected, all through the project lifespan, whenever the need for it was identified. Within this category we include mainly technical data (i.e., server load, client performance, lines of code) and specific real data (i.e., number and kind of most performed operations, amount of business objects handled).

The author of this thesis, as one of the few people that has been, and continues to be, part of the ARMISTICE development team since it was started in 2002/2003, has played an active part in such data gathering and interpretation, too. First as developer, as analyst later on, and nowadays as project manager, she has a deep knowledge and understanding about the system, the business domain, the design decisions, and the implementation details. The ARMISTICE development team has been formed by a non-static group of people, with up to five developers (three of them full-time) when the project was at its highest level of activity. Overall, the development has taken 200 person-months and the total code size is $\sim$ 83000 lines of code (LOC) for the server, and $\sim$ 66000 LOC for the client. Maintenance and bug-fixing take nowadays around 500 person-hours a year.

4

From requirements to analysis and design

As we saw in Chapter 2, all software development methodologies include as initial stage in their life cycles one or several tasks devoted to project requirements analysis. This involves conceptual evaluation and determination of system functionalities, extensive understanding of business domain, and knowledge extraction about business processes and activities.

After that, no matter whether the process to follow is accomplished adopting a sequential, iterative, incremental, prototyped, or agile approach, all the previous information is to be thoroughly studied and organised to produce a system design, which will be later implemented as a software product. Of great importance and relevance in this transition is the use of modelling artifacts such as software design patterns, and standard tools to capture and represent design information such as UML.

Even though we must of course do it for every software product we develop, when we deal with really complex and rather unknown domains, we would like to place special attention on the results of the analysis and design phases. The most expensive software problems to solve are those derived from bad requisite interpretation or design flaws, so no additional effort to ensure quality results from these engineering procedures will be in vain. Moreover, if we could formalise the output from this stage in a way that we could later use it during testing, we would be providing very valuable reassurance and verification traceable criteria.

In the following sections we review the key aspects of software requirements study, system analysis, and program design. Afterwards, our proposal to improve the confidence on the outlined solution, especially when dealing with complex problems and domains (such as risk management, our case study), is given: the formalisation of the extracted system concepts in a series of statements and equations which will allow further reasoning, properties extraction which will prove useful during testing, and a somewhat greater certainty and stronger confidence in our analysis and design activities.

4.1 Requirements elicitation

In software engineering, a *requirement* is an expression of desired behaviour or functionality for a software product or system. Requirements need to be defined, as a list of user's needs and wishes, and then specified, to link them to how the system to be built shall behave.

More and more frequently, software engineers find themselves facing the development of complex systems to address problems on business domains that they are not familiar with, and whose properties and problems they are not aware of or do not understand. In this regard, software engineering is one of the most challenging engineering disciplines, since a great part of the problem description comes from people's demands and preferences, which have a great subjective component, and not from neutral and objective factual data, such as slope inclination and terrain composition, or wavelength and transmission distances. In other words, software engineering has an important part of *social engineering*, as it needs to capture and understand people's problems first, to then find an efficient and quality solution for them.

A number of requirements elicitation or extraction techniques have been developed to deal with this first and essential stage in software construction, the most basic but effective one being **interviews** [69]. Other techniques, such as direct observation, task analysis, or simulations, can also be applied in some cases [70]. However, in spite of all given recommendations, requirements extraction is certainly an activity that requires more effort than is usually taken for granted. Most of the time, users have a hard time verbalising their needs, at least in terms that are straightforward useful for, or helpful to, the software engineers and developers. So, in the end, the success of this activity is highly dependent on the abilities of the people in charge. The essential aptitudes include good communication and social skills, such as effective use of language (both written and spoken), the ability to represent own ideas schematically and to interpret others' ideas, open-mindedness, decision-making competence, and even conflict resolution abilities; not to forget, of course, about application domain, technologies, and system programming knowledge. Obviously, all these attributes are hardly ever found in just one person, which is the reason why multidisciplinary teams are necessary, more than anywhere else, at this stage of the project.

4.2 System analysis

As a result of the requirements identification process, engineers should be able to outline the architecture of the system, based on the analysis of the technical requisites that are inferred from the description of functionalities and services needs. Using their technical expertise and previous experience, they must evaluate different options such as the most suitable kind of application (stand-alone, client-server, web. . .), the operative environment and hardware needs, as well as parameters such as concurrency, efficiency, or real-time behaviour demands. Using the elicited information, decisions concerning a wide range of variables must be taken: from the software development methodology that would be applied to the programming paradigm, and also the structure of the software to be built.

When facing the development of a new project, unless there are any unavoidable constraints that oblige, the use of standard technologies and notations is preferred. Not only because they help to build products which are easier to manage, evolve, and maintain [71], but also because by applying well-established norms or technical solutions the chances of successfully communicating and integrating the new product with already existing environments and systems will be greater. In particular, in the field of software analysis and design, standard modelling tools facilitate information representation, management, and interchange.

One of the most widely known modelling languages is the **Unified Modeling Language** (UML), a standardised general-purpose set of graphical notation artifacts to represent abstracts models of software systems [72]. Because of its properties and also thanks to the relative simplicity of the notation, UML constitutes a great resource. It serves not only as documentation and information exchange means within a development project, but also as a communication tool with the final user.

4.3 Program design

As we have already defined in previous chapters, software engineering is a computer science discipline devoted to the systematic and disciplined analysis, design, development, operation, and maintenance of software [1, 26]. It involves knowledge about methods and tools for defining software systems requirements, and also knowledge about tools and methods for designing software that fulfils those identified requirements, and for building, testing, and maintaining it. The theoretical principles that allow a group of software engineers to analyse complex fields (like risk management, for instance) and design a valid solution for their needs, are those from software analysis and design [73–76].

The central and fundamental instrument of software engineering to gather the essence of a problem, leaving all specific constraint details behind and reaching the main properties of a generic scenario, is *abstraction*. The software engineer moves from the actual needs to the formal definition of the main requirements, identifying the core parts of the software solution in the process, and then refining each component's task and goal in the system to be built. This activity, this creative process of transforming a problem into a solution, is what we call *design*. When referring to software design, it can be divided in two different stages: *conceptual* or system design (which is actually the output of the system analysis activity), and *technical* or program design.

Technical design can follow different paradigms (functional, object-oriented, domain-specific, process control...); there is also a large set of tools and notations that can be used to both document and convey the design of a software product (ER diagrams, UML diagrams, Petri Nets, data flow diagrams...). In any case, a good technical design will generally be based on a multi-component structure, promoting element independence for the sake of software reuse, which is a factor of quality as well as fault prevention and tolerance [77]. By keeping each system component both as a working element on its own and as an essential piece of the software gear (modularity), a versatile and robust system architecture can be more easily outlined.

The use of **software design patterns** is a key factor to carry out this task, ensuring the result to be efficient, flexible, and sound [78]. A design pattern is a repeatable software solution to a particular kind of problem, which has proved to be both efficient and simple [74]. Effective software design should anticipate and avoid potential problems that may appear later on, during the implementation phase. The use of design patterns has been shown an excellent strategy when it comes to both recognition and prevention of such subtle weak points. Besides describing a tested solution to a common scenario, they also represent a well understood way of interaction and communication among developers, improving readability of design documents, and even comprehensibility of derived source code in subsequent stages. Applying software design patterns during a system's design and development process helps to reduce the development effort, prevents the appearance of common errors, and guarantees the usage of already successful solutions.

4.4 Formalisation and model review

Faults or misconceptions in software product requirements extraction, system analysis, or design, are the most expensive problems to solve in a software project, unless they are detected before proceeding to the next step in the development (i.e., implementation and testing) [79]. To prevent them from occurring, software engineers and developer teams need to make sure they have properly understood users' requirements, and also that their analysis and

design work reflects and fulfils those requisites in the most accurate way possible. This is usually attempted by extending or complementing the acquisition interviews with *reviewing sessions*. Some iterative software methodologies already foresaw this need, and most of the development life cycles used nowadays schedule some room for this early reassurance activities. The specific way they approach it, however, differs: from the old prototype building idea to innovative simulation experiences, options have to be chosen depending on the particular constraints of each project and product.

Prototyping may work well in many cases, but potential differences between the primitive model and the final version of the software need to be kept in mind by all parties. Also, there are scenarios where this technique is not applicable, either because it is too expensive or due to other limitations, like the complexity of the business logic or the specificity of the underlying processes and algorithms. In such situations, only the help of diagramming tools and representation artifacts (such as UML diagrams) can help technicians and clients communicate over a product model.

Be that as it may, we present here a contribution to this stage of a development project, aimed to provide further confidence in the partial results of software analysis and design. Especially when dealing with critical systems or complex domains, we suggest to review the whole modelling process and to formalise the identified concepts and their interrelationships (cf. Fig. 4.1). Maintaining the high level of abstraction achieved during requirements, domain and system study, the objective is to write down semi-formal descriptive statements that characterise the conceptualisation of the business elements and the system components. Reading back, not only these equation-like sentences, but also the derived links and/or properties that can be extracted or inferred from them (adding and completing our view of the project specifications), to the user or client can be an additional way of verifying the accuracy of the work that has been done so far.

Besides, having a semi-formal description of a domain abstraction for a system can be of great help in later stages of the project, in particular if a declarative paradigm is chosen for the implementation task. Obtaining the properties the system to be built is required to present as output of this development stage, along with the rest of the technical details, means we provide not only the instructions for the implementation task, but also the features to look for at the validation stage. This link, that is hence established early between user requirements and system properties, is very interesting from the point of view of traceability and validation. We come back to this idea in Chapters 5 and 6.

4.5 ARMISTICE's business domain study

In Chapter 3, we presented our case study application ARMISTICE. In this case study, the methodologies we have described in the previous section have

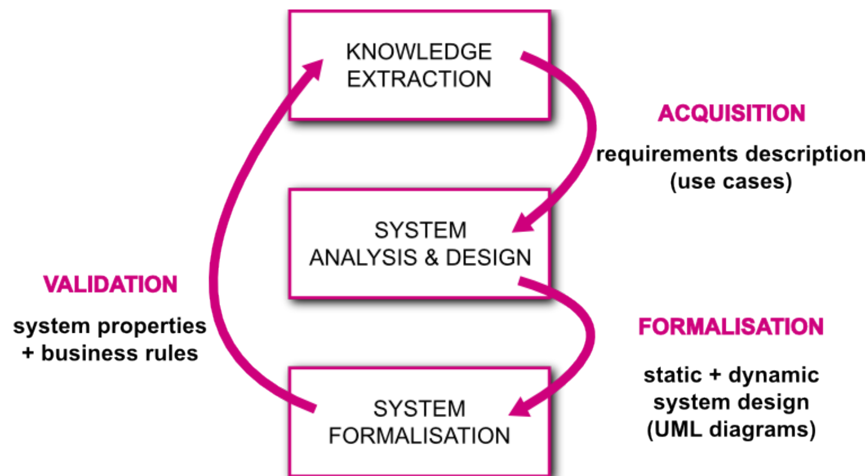


FIGURE 4.1. Knowledge elicitation and validation process

been applied: requirements elicitation, problem abstraction, and usage of software patterns to obtain a design of the functional architecture for the target software system.

4.5.1 Knowledge extraction

The methodology followed in the development of the ARMISTICE project varied from the first stages, when the stress was put on frequent meetings with the domain experts, to the final stages, when few meetings were necessary and the focus was on application testing. The initial appointments included long discussion sessions about both what the user demanded and what we, the technology experts, thought possible to achieve.

Risk management is not a well-known domain among computer scientists and software engineers, so many explanations, instructions, and clarifications were needed, especially during the first months. Once the essence of the domain and its main concepts were clear to us, the system modelling process started. For that matter, we found UML a very useful tool to communicate with the domain experts. As we depicted the system analysis and design in the shape of UML diagrams (mainly structure diagrams, but also behaviour and interaction diagrams), we showed and explained them to the experts in the field. After some introduction to this standard modelling language, they were quite easily able to understand them and soon even to make corrections and put their fingers on errors and misconceptions.

However, the properties of this case study, its intricate domain and the complexity of the software which was being built, demanded a more structured formalisation of the elicited requisites. This formalisation served both to establish a good level of confidence on the coherence of the elicited concepts,

and a model and properties to be used later on, to test the correctness of the system which would implement them.

4.5.2 Domain analysis and design

Risk management field and needs analysis shed light on the fact that regardless of the specific type of risk to face, the resources or processes exposed to that specific risk, the shape the threat might take, the different consequences it might have, in general, the approach in dealing with risks is common in all possible scenarios. This generalisation has been the key concept behind ARMISTICE.

When modelling ARMISTICE's business domain, we have gone one step further than usual, defining *meta-information* instead of just domain-specific information. Hence, the new system is a means of establishing which are the objects of interest, also referred to as *risk situations*, and the hazards threatening those risk situations, in the first place. But the way this is achieved is not only by introducing information about those risk situations in the system, but by previously establishing the related meta-information, i.e., the information about which *kind* of risk situations will the application manage. For example, a user interested in the risks affecting employees should be able to input the specific information about all personnel. But prior to doing this, the meta-information about those risk situations (employees) needs to be created, that is, the application will allow the specification of the 'person'-*type* object of interest, whose important properties can also be decided, for instance name, age, gender, qualification, job, salary, etc. This first high-level abstraction is called *risk group*.

In Figure 4.2 we present these concepts as a simple UML diagram[1], where each business object is represented by a square box, and relationships between them by arrows. Directed links reflect visibility properties, and multiplicity is also displayed on the diagram (default is one, $*$ means many).

[1] All through this section, some information and supplementary restrictions on the business objects are omitted for the sake of clarity.

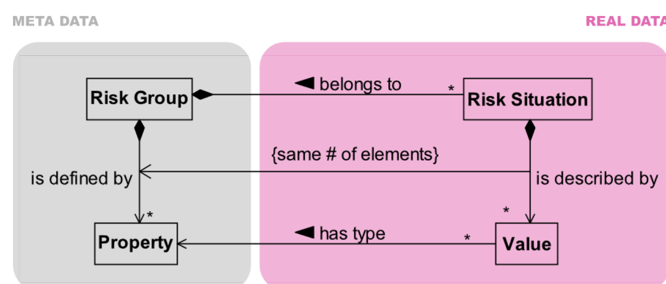


FIGURE 4.2. UML model of risk groups and risk situations

So, the diagram shows that every specific object of interest (i.e., every person in the staff) will be a *meta-instance* of a risk group, which specifies the relevant properties of a set of risk situations of the same type. After introducing this

meta-information, the user can proceed and input the necessary specific data about the employees.

Using this abstraction, it is possible to model the state of every insurable element. However, business requirements also demanded *transient* management of the risk situations. Based on Fowler's temporal patterns [80], tracking of the objects of interest is performed through time. The evolution of a specific risk situation is modelled since its creation as a set of *versions* and *revisions* (two-dimensional temporal modelling). A *version* represents a new state of an element which is meaningful for the business logic (e.g., a modification that may affect coverage). However, a *revision* represents a new state of the element which is meaningless as far as the business logic is concerned (e.g., modification to correct typos on the risk situation information). Figure 4.3 shows the UML representation of this feature, which is also applied to a number of other elements in ARMISTICE (those for which it is relevant to keep track of changes).

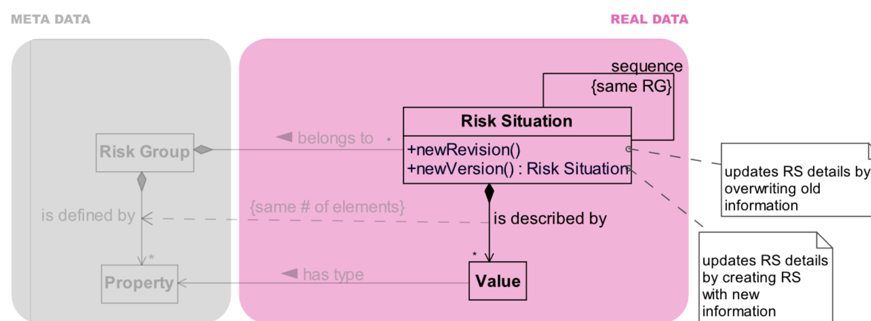


FIGURE 4.3. UML model of risk situation dynamism

A similar analysis and design process involves *hazards* threatening the risk situations: if the objects of interest are people, the main hazards may be long-lasting illnesses or strikes; if the objects of interest are warehouses or offices, the relevant hazards to be taken into account by the system may be arson, flooding, or theft. Again, it is the expert user who will first decide what the meaningful risks to the business area are, and then introduce them into the system, classifying them (if applicable) under the appropriate categories (cf. Fig. 4.4).

Once risk situations and hazards have been created in the system, *exposures* are set to match pairs of risk situations and hazards potentially affecting them (cf. Fig. 4.5). Many different types of objects of interest can be registered in the system, as well as many different hazards. However, not all hazards threaten the same kind of risk situations. A person is probably not vulnerable to theft (unless she performs an important "intellectual" role and competitors may be interested in head-hunting or recruiting valuable employees), but the contents of a warehouse certainly are. Thus, exposures represent the infor-

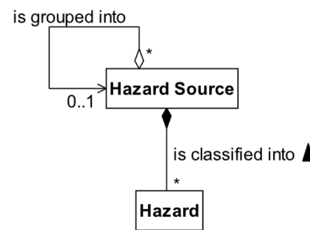


FIGURE 4.4. UML model of risk hazards

mation about which hazards we want to bear in mind when referring to certain risk situations. When a exposure link is established between a hazard and a risk situation, some interesting values are assigned: probable maximum loss (PML), estimated maximum loss (EML), normal loss expectancy (NLE), intensity, and frequency.

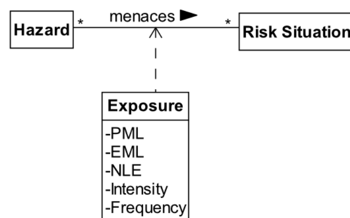


FIGURE 4.5. UML model of exposures to risk

Exposure information is used by system experts to ensure that, ideally, every risk situation is protected against all hazards it might be exposed to. In any case, regardless of the risk management policy alternative which is being applied, this is a means for the responsible person of being aware of the specific menaces that threaten every resource of the organisation.

These are the first system configuration steps ARMISTICE needs, so it will contain the basic business data to start working with. But the goal is much more than to build just a risk situations or hazards database: it is to build a complete tool to assist in the implementation of a company's overall risk management policy. To do so, the next important functionality is the management of *insurance policies*, detailed down to the level of insurance warranties, which specify the terms of the contracted coverage for a set of risk situations, when some specific conditions become present. Furthermore, at the warranty level, the system will also deal with the formulae which calculate excesses and limits when the transference is not total.

Insurance policies are the most complex element in risk management, and consequently also in our system. These formal documents, pages and pages long, detail all the norms, rules, and regulations previously agreed upon by the

parties: not only which specific objects are being considered or which particular hazards are being taken into account, but also relevant dates (when the agreement validity commences and when it expires) and all sorts of applicability conditions and constraints they decide upon. Once an insurance policy term comes to an end, the agreement can be renewed as is, or it can be modified to include subtle variations, or it may be renegotiated from scratch. Of course, changes can also be made by mutual consent even during the policy validity period, meaning an amended document or addendum will be written down, where the new terms and conditions will be put on record and be in effect at that very moment. Depending on the business area, this kind of modifications may even be foreseeable, so that the new terms applicable, if they finally appear, can be stated and agreed upon in advance.

To fulfil this real-life behaviour of a policy life-cycle, ARMISTICE has been designed to allow modelling of insurance policies as a set of *renewals* (cf. Fig. 4.6). A renewal represents a new policy created to provide coverage to a set of risk situations over a certain time interval. At the same time, a renewal can be broken down into one or more *supplements* (endorsements). A supplement represents a revision of the policy, made to establish or modify its coverage, its contractual clauses, etc.

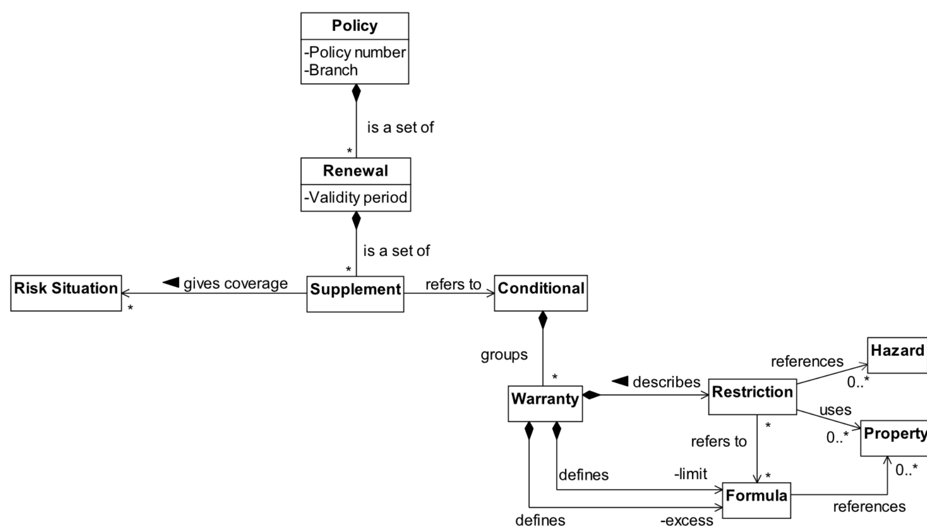


FIGURE 4.6. UML model of insurance policies

Apart from some indispensable information such as the set of covered risk situations, the relevant dates, and other attributes (such as different sorts of general limits and excesses), the essential core element of a supplement is the *conditional*. Conditionals model the constraints under which an insurance policy provides coverage for a claim. In other words, the supplement conditional is a model of the contractual clauses of a specific policy, the model of the policy coverage, that is to say, the model of the policy *warranties*.

As descriptions of legal terms or insurance constraints, conditional *restrictions* must allow the inclusion of references to actual risks, properties of the risks situations that are being covered, as well as other policy-related data and calculations (*formulae* for limits, excesses, etc.). As we will see in the next chapter, these models of policy coverage will also include short explanatory details (nuances) in natural language, which will allow ARMISTICE to very faithfully represent reality, and make it possible to obtain extremely accurate results when automatically suggesting the most appropriate policy to charge with the expenses of an accident.

We have to remark that formulae, restrictions, and even conditionals, are modelled as separated elements of its own for the sake of re-usability. The way excesses or limits are calculated, the kind of conditions that are checked to determine a clause's applicability, and even the combination of both, are very likely to be similar or even just the same in different sets of policies. So modelling each of these elements as full and independent entities allows the user to build her own "library" of these components and then use them as many times as needed, thus saving, not only a lot of time and effort, but also potential mistakes.

Therefore, ARMISTICE combines the information vs. meta-information distinction at the business-data level with the translation of extremely complex real elements such as policies into objects which can be handled by a computer. Once all this data and logical structures are in the system, it should be ready to assist risk managers. First, to manage accidents as soon as they occur, to decide which of the contracted applicable policies is the most suitable or desirable to apply in each case, and then to be aware of the life of the claim, tracking the accident from the starting point until the file is closed. Second, to analyse all data and make decisions about the suitability of the current risk management policy that is being put into practise, as previously mentioned.

The accident *claim* management process involves becoming aware of the risk situation(s) that has (have) been affected by a particular hazard, *estimating* losses and repairs costs, tracking all related *tasks* and activities needed to repair the damages, *payment* issuing and processing, *indemnities* claiming and recovery,... This data helps the system to keep the claim status up-to-date, right through and up to the final stage when everything is solved and the file is finally and permanently closed. A UML description of this last scenario is showed in Figure 4.7.

But even then, ARMISTICE's functionalities will not be over. Apart from these everyday kind of operations, there is potential for analysis that can be performed on the basis of all the information gathered daily. At any time, the system can be queried for information to show if the risk management policy is being successful, i.e., if the losses are being recovered as desired, if any of the contracted policies are redundant or superfluous, if there is any hazard causing uncovered accidents because it was missed or underestimated at in-

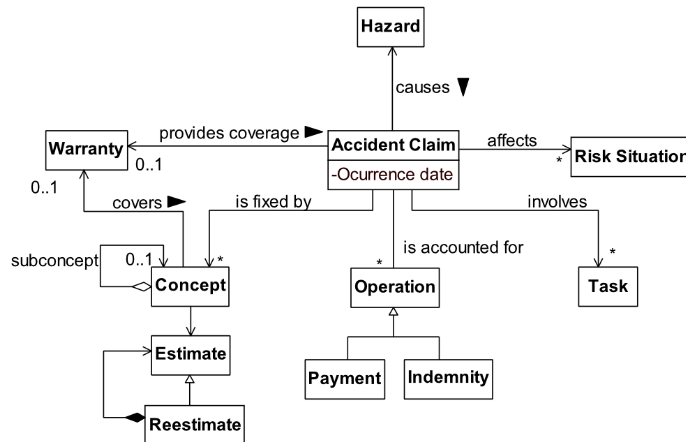


FIGURE 4.7. UML model of accident claims

surance negotiation time. This task is not to be underestimated, because it can help to detect deviations on the risk management policy at relatively early stages and, hence, to correct them.

4.5.3 System formalisation

In this last section of the chapter, the formalisation of ARMISTICE's essential concepts is explained. This is our suggested means of increasing confidence in proper knowledge acquisition, by pursuing the validation of the main abstractions acquired during the design stage. This also helps to improve and fine-tune the acquired knowledge, as well as to anticipate the formulation of testing requisites as directions for later test case generation.

Consequently with the discussion sequence followed for the domain analysis, we start by formalising the definition of a *risk situation* (RS) as an element which models the state of any object of our interest (e.g., a person, a shop, or a vehicle). Every RS is also an instance of a *risk group* (RG). An RG is a template, that is, a meta-description of all the important attributes (e.g., name, age, address, vehicle registration number, warehouse area, or cubic capacity) which describe the state of any relevant element from the risk management point of view (e.g., employees, commercial premises, or industrial vehicles).

We denote the set of all possible attributes as $\mathcal{A}$, each of them defined as

$$Attribute \quad = \quad (name, type)$$

where

$$\begin{aligned} name &\in String \\ type \in Types &= \{String, Boolean, Timestamp, Money, \ldots\} \end{aligned}$$

Over these elements, the set of risk groups $\mathcal{RG}$ is defined. Each member $\mathcal{RG}_i \in \mathcal{RG}$ is again a set, as follows:

$$\mathcal{RG}_i = \{a_0, a_1, \ldots a_{n-1}\} \quad / \quad \forall x \in [0, n-1],\ a_x \in \mathcal{A} \tag{4.1}$$

Therefore, an RG is a set of n attributes which provides a meta-description of a kind of RS that is important from the user's perspective.

Using the $\mathcal{RG}$ superset, we can define the set of risk situations $\mathcal{RS}$; a RS $\mathcal{RS}_j$ is an instance of an RG $\mathcal{RG}_i$,

$$\mathcal{RS}_j : \mathcal{RG}_i \longrightarrow \cup\, Types$$

such that every attribute in the RG has a concrete value assigned in the RS:

$$\mathcal{RS}_j = \{p_0, p_1, \ldots p_{n-1}\} \quad / \quad \forall x \in [0, n-1] \qquad \begin{aligned} & p_x = (a, v) \\ \wedge \quad & a \in \mathcal{RG}_i \\ \wedge \quad & v \in Type(a) \end{aligned} \tag{4.2}$$

Hence, an RS represents indeed the state of a specific element of interest. For short, we will use $\mathcal{RS}_j.a$ to denote the value of attribute a in the RS $\mathcal{RS}_j$.

Temporal tracking of the elements in $\mathcal{RS}$ is performed by slightly modifying the original definition of an RS as follows:

$$
\begin{aligned}
\mathcal{RS}_j = \{\mathcal{RS}_j^{v_0}, \dots \mathcal{RS}_j^{v_{\alpha-1}}\} \quad / \quad & \forall x \in [0, \alpha - 1] \qquad v_x \in Timestamp \\
& \wedge \quad v_x < v_{x+1} \\
& \wedge \quad \mathcal{RS}_j^{v_x} = \{\mathcal{RS}_j^{v_x, r_0}, \dots \mathcal{RS}_j^{v_x, r_{\beta-1}}\} \\
& \wedge \quad \forall y \in [0, \beta - 1] \quad r_y \in Timestamp \\
& \wedge \quad r_0 = v_x \leq r_y < r_{y+1} \\
& \wedge \quad \mathcal{RS}_j^{v_x, r_y} = \{p_0, p_1, \dots p_{n-1}\} \\
& \wedge \quad \forall z \in [0, n - 1] \quad p_z = (a, v) \\
& \wedge \quad a \in \mathcal{RG}_i \\
& \wedge \quad v \in Type(a)
\end{aligned}
\tag{4.3}
$$

So the elements belonging to the $\mathcal{RS}$ set can be defined as a collection of items representing different states of RSs:

$$
\begin{aligned}
\mathcal{RS} = \{\mathcal{RS}_0^{v_0, r_0}, \dots, \mathcal{RS}_j^{v_x, r_y}, \dots\} \;/\; \forall j \in [0, |RS| - 1] \quad & x \in [0, \alpha_j - 1] \\
\wedge \quad & y \in [0, \beta_{j,x} - 1]
\end{aligned}
\tag{4.4}
$$

where $|RS|$ is the actual number of different modelled RSs. The expression $\mathcal{RS}_j.a$ must be changed to $\mathcal{RS}_j^{[vDate][rDate]}.a$ where $vDate, rDate \in Timestamp$.

With this new definition for temporal management, complex questions like "At the moment in time $rDate$, what was the value we *thought* attribute a (of RS $\mathcal{RS}_j$) had at date $vDate$, and which one we do *know* it is now?" can be answered. Figure 4.8 shows a schematic view of the evolution of the versions and revisions of a RS $\mathcal{RS}_j$ through time. In particular, the evolution of the RS state in the time interval $[v1, v2]$ is highlighted. Here the modification is caused by the correction of a hypothetical mistake: before temporal point $r1 > rDate$, the value of a (number of employees) in the RS was 7; but after $r1$ it was updated to 8.

A similar formalisation process involves hazards threatening the risk situations. We call

$$
\mathcal{H} \quad = \quad \{h_0, h_1, \dots h_{n-1}\} \quad / \quad \forall x \in [0, n - 1] \tag{4.5}
$$

where each h_i is to be defined by the ARMISTICE user (e.g., fire, explosion, or terrorism). A hazard $h_i \in \mathcal{H}$ can act over an RS causing a damage or accident.

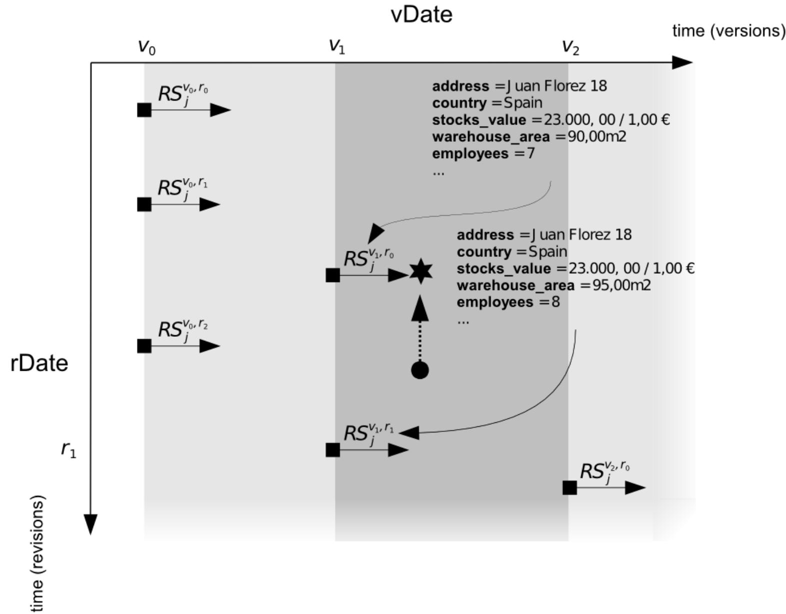


FIGURE 4.8. Versions and revisions of a risk situation

Respectively, an exposure $e_i \in \mathcal{E}$ is a quantified relationship between a risk situation and a hazard,

$$e_i : \mathcal{RS}_j \times h_k \longrightarrow (PML, EML, NLE, i, f) \tag{4.6}$$

The set of relationships $\mathcal{E}$ represents, as we previously said, the information about which combination of RSs and hazards may be involved in an accident. In other words, it is the probability of a damage on a specific RS caused by the presence of a specific hazard. Such a probability is expressed by the associated factors:

- *Normal Loss Expectancy*, $NLE \in Percentage$, a measure of loss-recovery costs under normal conditions.
- *Estimated Maximum Loss*, $EML \in Percentage$, a measure of recovery costs which assumes the existence of impairments at the time of loss.
- *Probable Maximum Loss*, $PML \in Percentage$, an estimate of the largest loss that is likely to be suffered.
- *Intensity*, $i \in Degree$, a qualitative measure expressing how seriously the hazard can be expected to manifest itself.
- *Frequency*, $f \in Degree$, a qualitative estimate of the number of occur-

rences an accident affecting the RS due to the hazard under consideration can be expected within a specific time period.

where $Percentage \in Types$ and $Degree \in (Low, Medium, High)$. This additional information (PML, EML, NLE, intensity, and frequency of an exposure) will be very valuable and useful for the analysis of the risk management measures and their effectiveness.

Carrying on the formalisation to the next level, ARMISTICE works over a set of policies $\mathcal{P}$, where each policy $\mathcal{P}_i$ is modelled as a set of renewals $\mathcal{P}_i^{rj}$. At the same time, a renewal is composed by a set of supplements $\mathcal{P}_i^{r_j,s_k}$:

$$\begin{aligned} \mathcal{P}_i &= (Number, Branch, \{\mathcal{P}_i^{r_0}, \dots \mathcal{P}_i^{r_{\alpha-1}}\}) \\ \mathcal{P}_i^{r_j} &= (ValidityStart, ValidityFinish, \{\mathcal{P}_i^{r_j,s_0}, \dots \mathcal{P}_i^{r_j,s_{\beta-1}}\}) \\ \mathcal{P}_i^{r_j,s_k} &= (Coverage, Conditional) \end{aligned} \quad (4.7)$$

where $Number, Branch \in String$, and $ValidityStart, ValidityFinish \in Timestamp$ are the dates used to determine the validity period of the renewal. As for the $Coverage$, it represents the collection of RSs building the set of objects insured by the policy. Of course, every $\mathcal{RS}_j$ inside the $Coverage$ set is different, and each of them refers to a particular version of the RS:

$$\begin{aligned} Coverage \subseteq \mathcal{RS} \quad / \quad & \forall\, \mathcal{RS}_{j_1}, \mathcal{RS}_{j_2} \in Coverage \Rightarrow j_1 \neq j_2 \\ \wedge \quad & \forall\, \mathcal{RS}_{j_i} \in Coverage \Rightarrow \mathcal{RS}_{j_i} = \{\mathcal{RS}_{j_i}^{v_x,r_y}\} \in \mathcal{RS}_j \end{aligned} \quad (4.8)$$

Regarding the $Conditional$, $\mathcal{C}_i \in \mathcal{C}$, it is used when looking for coverage for a claim, and can be represented as

$$\mathcal{C}_i = \{g_0, \dots g_n\} \quad / \quad \forall x \in [0, n],\ g_x = (Restriction, Excess, Limit) \quad (4.9)$$

each warranty g_x containing an applicability precondition $Restriction \in \mathcal{R}es$ and some formulae $Excess$, $Limit \in \mathcal{F}or$, used to calculate the corresponding excess and limit.

Each element $\mathcal{F}or_i \in \mathcal{F}or$ is a formula which models a calculation:

$$\mathcal{F}or_i \ : \ \{p_j \ / \ p_j = (a, v) \wedge a \in \mathcal{A} \wedge v \in Type(a)\} \longrightarrow \cup\, Types \quad (4.10)$$

where the p_j operands are either user-defined values or properties which may be present on risk situations. Besides, excesses and limits need to have monetary values, so actually $Excess, Limit : \mathcal{F}or_i \longrightarrow Money$.

On the other hand, restrictions are the conditions to be held by a warranty in order to be activated (i.e., determined as applicable) when an accident occurs. Thus, every warranty (i.e., contractual clause) inside a conditional of a policy is an applicability constraint which models its behaviour. Formally speaking, each element $\mathcal{R}es_i \in \mathcal{R}es$ is an expression which may involve user-defined values, risk properties, hazards, and even literals ($s \in String$):

$$\begin{aligned}\mathcal{R}es_i : \{p_j/p_j = (a,v) \wedge a \in \mathcal{A} \wedge v \in Type(a)\} & \\ \times \quad \{h_k \;/\; h_k \in \mathcal{H}\} & \\ \times \quad \{s_m \;/\; s_m \in String\} & \quad \rightarrow \quad \{Boolean \cup String\} \\ & \quad\quad\;\; \times Integer\end{aligned} \tag{4.11}$$

When a restriction is evaluated, it can turn out to be $true$ or $false$ (i.e., the associated warranty can be applied or not). However, due to the inclusion of human-language nuances $s_m \in String$, sometimes it will not be possible for the system to state directly whether the restriction is true or not. Due to the unpredictability of these nuances s_i (and for simplicity reasons), which are specified by the human user to reflect any additional constraint or detail, they have not been formalised. Had they been so, natural-language processing techniques [81] could have been used, for instance. In our case, however, the truth value of a restriction will depend on the answer of the user to the terms expressed by those manually described conditions included in the restriction (in other words, it will only be evaluated to a boolean value by means of human user intervention). Every evaluated restriction has also an associated *weight* expressed as an $Integer$ value which is used as sorting criterion among the contractual clauses which could provide coverage against an accident (if more than one should be found).

Finally, we will take a look at the basis of claims. In a formal way, we can describe the set of *claims* in the system, $\mathcal{CM}$, as a set of elements

$$
\begin{aligned}
\mathcal{CM}_i = (ODate, h_k, \mathcal{ARS}, g_x) \quad / \quad & ODate \in Timestamp \\
\wedge \quad & h_k \in \mathcal{H} \\
\wedge \quad & \mathcal{ARS} \subseteq \mathcal{RS} \\
\wedge \quad & \forall\, \mathcal{RS}_{j_1}, \mathcal{RS}_{j_2} \in \mathcal{ARS},\ j_1 \neq j_2 \\
\wedge \quad & \forall\, \mathcal{RS}_{j_i} \in \mathcal{ARS},\ \mathcal{RS}_{j_i} = \{\mathcal{RS}_{j_i}^{v_x, r_y}\} \in \mathcal{RS}_j \\
\wedge \quad & g_x \in \mathcal{C}_i\ ,\ \mathcal{C}_i \in \mathcal{C} \\
\wedge \quad & \mathcal{C}_i \in P_x^{r_z, s_y} \\
\wedge \quad & P_x^{r_z}.ValidityStart < ODate < P_x^{r_z}.ValidityFinish \\
\wedge \quad & \mathcal{ARS} \subseteq P_x^{r_z, s_y}.Coverage \\
\wedge \quad & g_x.Restriction \equiv true
\end{aligned}
\tag{4.12}
$$

that represents the relationship between the RS affected ($\mathcal{ARS}$) by a hazard (h_k) on certain date ($ODate$), and the warranty that provides coverage for that specific accident (g_x). $ODate$ is the date used to find the applicable policy renewals (i.e., those whose validity period has already started but has not expired yet) for which supplements the set $\mathcal{ARS}$ is found to be part of their $Coverage$. Once supplements appropriate by date and coverage are identified, their warranties must be examined, to determine the ones with positive restriction evaluation. Among those, the most suitable warranty to cover the claim will be chosen to be charged with all the loss recovery expenses, either by the system (if the choice is automatically decidable) or by the user.

As Equation 4.12 shows, the same comment we made when talking about policies $Coverage$ (cf. Eq. 4.8) is applicable here: each $\mathcal{RS}_j$ inside the $\mathcal{ARS}$ set must be different. As both sets will be checked against each other for total/partial matches, this makes sense.

4.6 Formalisation benefits: validation planning

The structure of the abstract concepts and their relationships produced by the formalisation of the analysis and design constitute a model of the system elemental components and properties. The formalisation process we have carried out is extremely helpful in assuring that a good analysis and design has been performed, and the methodical and explicit description it provides will be even more useful in upcoming stages of the project.

First of all, the results of the formalisation can be presented to the users for validation. They constitute a different way of expressing system properties and business entities characteristics, a reformulation of requirements usually stated in natural language, or depicted in UML diagrams. Any alternative representation we use forces us to describe our application requisites in a different

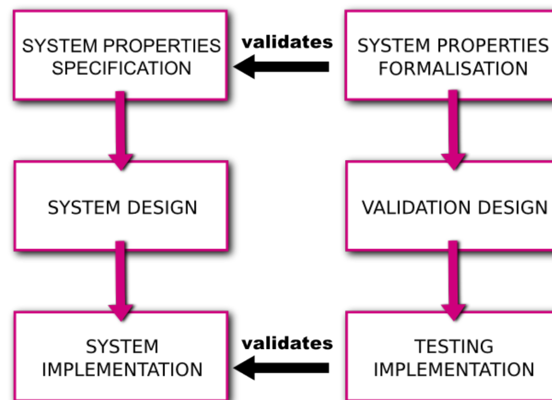


FIGURE 4.9. Correspondence between development stages and validation activities

style, using a distinct set of conventions which have different expressive capabilities. This exercise is more likely to reveal ambiguities or under-specified details than using just a single way of reporting and documenting. Thus, it increases the chances of identifying weak or flaw specification spots as soon as possible, which is very convenient as was pointed out in this chapter.

Secondly, this formal description of properties and features is a high-level representation, closer to human reasoning and also to the abstraction level of the programming paradigm we intend to use (i.e., functional programming). This reflects directly on the effort required for the implementation task, since the way of coding behaviour in declarative languages is based on the description of functionalities and the pattern-matching of data instead of the procedural dictation of instructions.

Last but definitely not least, the combination of these benefits opens the door to a planned-in-advance testing stage of both static and dynamic system properties. The static properties stated by our formalised equations are plain descriptions of business objects and their attributes. For instance, directly from Equations 4.1 and 4.2 (page 45), we can write down the following properties:

> "*Every risk situation object must have a number of properties which is equal to the number of properties of the risk group object it is associated with.*"

and

> "*Each property of a risk situation object must have the same type as one property in the risk group object it belongs to.*"

The properties can be translated, in a straightforward way, into data generators to be used with automatic testing tools for unit testing, as will be explained in Chapter 6. In a similar manner, the dynamic properties, the set of busi-

ness rules that affect different business objects and their behaviour, descriptive human-language statements, are usually established as global properties that need to hold and commonly only used to write test case samples. Again, in Chapter 6 a technique to automatically generate test cases from these business rules descriptions is presented, following the same philosophy.

5
Implementation of a paradigm shift

Analysis and design are meant to gradually take shape through software implementation. The real-world concepts and business domain requirements that were captured during previous development stages will finally be translated from UML diagrams and other specification documents to specific pieces of source code.

There is a wide range of factors that influence the choice of the most convenient implementation environment for a software project, including not only the programming language, but also any other auxiliary development tools, specific utility libraries, etc. Sometimes, the decision is bounded by unavoidable dependencies with some specific-purpose third-party software, it may be influenced by the development team technical skills (personal preferences or prejudices), it may be suggested by previously successful similar experiences, or it may be constrained due to other political or economical reasons.

In this chapter, however, we will assume that none of the aforementioned factors are in place, thus no preconditions or constraints are being put before us. That needs to be the case for introducing the paradigm shift this thesis advocates. With no external reasons affecting this election, the focus can be placed just on unbiased technological suitability arguments.

The following sections first briefly present the novelties, changes, and adaptations that appear in the implementation phase of a software product when the

functional paradigm is applied, with respect to more extended development approaches. Then, the benefits that come with the adoption of functional principles are explored. Formalisation of elicited analysis concepts as shown in Chapter 4 is justified now thanks to the usage of a functional language for the implementation. Program logic and behaviour are translated into a few lines of descriptive high-level code, saving time and efforts, producing a high-quality result, and enabling the software engineers to tackle more complex problems and scenarios.

5.1 Is such a change feasible?

The popularisation of modelling artefacts, such as software patterns and representation tools such as UML notation, has come hand in hand with the blooming of object-oriented languages such as Java or C++. The analysis and design activities conducted with the assistance of these abstractions and utilities are a perfect fit for the intrinsic properties of those languages (the concepts of class and object, inheritance, abstraction, encapsulation, polymorphism). In fact, this combination of procedures and outcomes forms what has been traditionally called *object-oriented development*.

However, while modelling real world concepts as entities (i.e., objects) with certain properties (either static –attributes– and/or dynamic –behaviour) is a quite natural approach to generic-purpose software analysis, it does not necessarily mean that it can only be followed by an implementation in an object-oriented language. As already stated in the previous chapter, the requirements elicitation and system design tasks can be clearly distinguished from program construction: the first two aim to determine the functional requisites of the system, and it is the last one which links them to the coding stage producing implementation specifications. In other words, only after stating *what* the application or system (at all levels) needs to do, we move to *how* it will do it.

The advantage that the object-oriented programming paradigm has claimed as fundamental, and the main explanation for its dominance over other paradigms, is based on the straightforward mapping of the concepts in an object-oriented analysis model onto implementation classes. In contrast, any alternative approaches would necessarily need to deal with the problem of translating such a conceptual model to a completely different set of implementation constraints and terminology.

But clearly, the previous diagnosis does not take into account other potentially relevant properties that different paradigms may present, apart from programming detail specifics. For that reason, it fails to recognise that even though there may be little room for discussion about how convenient it is to transfer object-oriented concepts to object-oriented source code, it need not be the case for such a step to be infeasible in other environments, though due to different reasons. Specifically, considering a functional approach, despite the

implementation details being far away from those of object-oriented programming, its descriptive nature and intrinsic higher level of abstraction places this alternative in a much closer position to that of concepts and definitions, which are the ultimate components of any design.

This is why this dissertation presents the functional paradigm as an option worth considering for the implementation phase of software development processes, no matter that the previous phases have been addressed using techniques which are commonly associated only with object-oriented programming. In addition, if we refer to the distributed functional paradigm, which provides transparent access to features such as concurrency, distribution, or persistence, the chances for improvement are even greater.

5.2 Translation of borrowed concepts

Object-orientation is an approach to software development that organises both problem and solution as a collection of discrete *objects*, an abstraction which blends together data structure and behaviour. Despite the properties, features, or reasons that may justify the additional effort of dealing with concepts from two different programming paradigms in a specific programming language or environment, there is of course a set of basic definitions and formalisms that need to be translated, unavoidably, when object-oriented analysis and design are to be applied in any non-object-oriented environment. These elementary aspects include, mainly, the object/class abstraction and the inter-relationships between objects.

This is an old challenge, though, that has been successfully addressed many times and from different perspectives, as we will see in the next subsection.

5.2.1 Object-orientation in non-object-oriented environments

The idea of applying object-oriented concepts and techniques in non-object-oriented environments is almost as old as object-orientation itself. From the moment this new paradigm broke into industry, different initiatives where put in practise in order to integrate the new software development philosophy with the already-existing procedures and organisational routines and schemes. The precursor efforts in this direction go back to the early nineties, where object-orientation was first tried to be applied in a C-programming environment.

Being one of the most popular imperative languages at the time (after displacing a whole generation of languages from the fifties, such as Fortran, Lisp, and COBOL, more than a decade before), the C language was the most common scenario for these initial integration approaches. The majority of them suggested that classes were implemented as structures, and associations between classes as pointers to structures. More complex constraints of object-orientation, which were more difficult to reproduce in a non-object-oriented

environment, were surpassed or mimicked in the best possible way (like inheritance, replaced by delegation), and the whole process involved a firm discipline (properties like encapsulation depending strongly on this) [82].

But even though this kind of projects served well for the specific purposes they were designed for, they were almost completely abandoned after some years, as a result of the extended and definite replacement of plain structured programming by object-oriented programming. A new step in the ongoing evolution of languages and software development techniques had taken place. And still, there were very successful experiences that have survived even until our days. This is the case of a very well-known international project involving a large community of users and developers: the GNOME project [83].

5.2.1.1 The GNOME project

Born in 1997, the GNOME project is an ongoing community development effort intended to build a complete, easy-to-use, accessible desktop environment for users and a powerful application framework for software developers. GNOME is a free software object-based desktop implemented in the imperative language C and distributed according to the GNU General Public License (GPL). The project is steered by the GNOME Foundation, which receives strategic guidance from worldwide software companies and organisations, such as Sun Microsystems, Nokia, Google, Intel, IBM, and the Free Software Foundation, among others.

While most GNOME libraries are written in plain C, all their graphical widgets are implemented as objects following the principles of object-oriented programming. The technical solution applied to fill in the gap between procedural and objectual programming follows the same line that has already been mentioned: *GObject*, the basic element of the fundamental *GLib* library, is a particular kind of C structure where specific fields are used to inherit state, and function pointers may resolve behaviour along a chain of inheritance. Even though enforcement of objective structuring is not as strict as it would be when using an object-oriented language, the use of standard coding practises and naming conventions has definitely benefited the project. The reasonable ease of use, together with advanced capabilities such as reference counting and event propagation, have confirmed polymorphic and extendible GObjects as a long-lasting successful experience.

Be that as it may, this mixture of technologies is certainly not common. Most companies have switched completely to object-orientation, or else stuck to traditional imperative languages for their own reasons. This, however, has not prevented the GNOME developers community to spread and grow broader. Both programming styles are popular enough among today's professionals (and even amateurs) not to be an obstacle to anyone who is interested in collaborating, as the numbers show [84, 85], regardless of having a background

as C programmers or coming from the object-oriented world. And this can only be an encouragement for this dissertation's objectives.

5.2.2 Object-orientation and functional environments

Similarly to what happened with procedural languages, declarative programming has not remained unaware of the benefits that came along with object-orientation. Several decades older than the latter, functional languages were nonetheless always regarded as academia experiments, not mature or efficient enough for the outside-world. It is true that their most famous representatives were, in fact, born as academia products (such as ML –University of Edinburgh– or Haskell –University of Glasgow) in contrast with industry-originated languages (such as FORTRAN –IBM, C –Bell Labs, or more recently Java –Sun Microsystems), with the exception of Erlang, which has been developed by Ericsson.

Yet, there are widely recognised multi-paradigm languages which combine object-oriented and functional concepts. That is the case of Objective Caml (OCaml), which is not only an example of the relevance and interest of less dogmatic approaches to software development, but more importantly, probably the main implementation of Caml, a dialect of ML created at INRIA. Developed in 1996, OCaml extends the Caml core with object-oriented constructs, adding an object layer to the original functional features such as pattern matching or first-class functions.

On the other hand, some authors have already pointed out that a purely object-oriented world view can constrain the achievement of appropriate solutions due to the incompleteness of its ground concepts, that therefore unavoidably restricts the available set of possibilities [86]. Multi-paradigm solutions are again suggested as a wise way of eluding this problem, and efforts to apply functional abstractions and patterns to enhance object-oriented design have been explored.

All these reason lead us to the thought that maybe it is time for a paradigm shift, or perhaps to steadily direct our efforts towards a serious multi-paradigm development style at last. The trend is out there already, the change is slowly starting. Even big software companies do not regard declarative programming and functional languages as academic tools any more. The most evident proof of this in some recent products: Microsoft F# [87], for instance, is a multi-paradigm language which supports object-orientation and also functional programming, a sort of OCaml dialect. Scala [88] is another language whose popularity has increased enormously in recent times, as it offers a smooth combination of object-oriented and functional languages features, together with seamless Java integration.

All the steps in the history of programming have taken us to broader, more abstract, higher-level ways of designing and implementing software. The natural

evolution is to proceed towards the most advanced of these choices: declarative programming.

5.3 Strengths of declarative languages

Declarative languages, and among them, functional ones, present a set of powerful and versatile features that have, from the beginning, succeeded in gathering the attention of academia all around the world. Efforts in translating that interest to industry have not had good results so far, but the maturity that has been reached and the particular development and operational needs that we are facing at the moment can be the proper scenario in which finally this valuable technology can be transferred from campuses to companies.

The main properties that, though not present in all declarative languages, are commonly associated with declarative programming, are:

Reduction semantics, computations as side-effect free reductions of expressions instead of instructions operating on an implicit state.

Higher-order functions, functions that can be both provided as argument to and returned as result from other functions.

Lazy evaluation, avoiding any computation until its actual value is indeed needed.

Pattern-matching, operation on data based on testing and/or deconstructing elements for values or according to a specification.

Type inference, using the surroundings of an expression to automatically compute the most general type expression for it.

In contrast with declarative programming, object-oriented programming, whose main abstraction are autonomous objects (representations of business-domain complete entities, captured in a taxonomy of inheritance relationships, that have self-supporting state, and provide self-sustained operations), is characterised by a set of properties that rarely appear in declarative environments:

Encapsulation, clear differentiation between an abstraction's structure and its behaviour.

Inheritance, incremental derivation of objects from other objects by *specialisation*.

Dynamic binding, run-time determination of the exact implementation of a request that will be executed.

Identity, object existence considered independent of its value or state.

The question of what is best to have among these two sets of characteristics does not have a definite answer. Software engineering is nowadays mature enough not to believe in, or look for, silver bullets any more. In some scenarios,

higher-order functions can make a difference, as can inheritance in others. Some problems can be enormously simplified by means of encapsulation, and others can be avoided if reduction semantics is in place. So the choice does not only involve picking one of these sets of properties, it includes employing the philosophy behind the paradigm that offers them.

The functional decomposition implied by declarative programming means that everything is seen, and thus modelled, as a function or set of functions. On the other hand, object-oriented decomposition establishes a correspondence between modelled objects and the real world. The object-oriented perspective is closer to the way we perceive reality (as identifiable things with state that change over time), but does not perform that well when carefully looking at the behavioural details. The functional perspective is not as good at the big scale of designing activity agents who carry out activities or implement services, but it works nicely in the small and specific, allowing to code complex behaviours in easy and safe ways. Objects grant greater stability of actors and interfaces which offer services, but do not prescribe order of actions, neither do they avoid the developer having to deal with their internal complexities. When using object-oriented analysis and design, object-oriented programming offers seamless development, but using "classes for everything" is not feasible nor desirable in all cases.

This dissertation aims to offer a third and integrating alternative, proving that the benefits of object-oriented analysis and design need not be incompatible with those of the functional programming properties.

5.3.1 Functional patterns

Similar to what happens with design patterns, there are many situations in which developers find themselves implementing similar scenarios over and over again. An analogous role to that of design patterns during the analysis and design stages is that of functional patterns at the implementation stage.

Patterns in functional programming have been studied in depth by a number of authors, who have identified the most common statement sequences and behavioural templates to be found in functional source code, at different levels of abstraction, from the most essential strategic programming [89] to high-level application tasks [90]. This, in addition to the extensive research that has been conducted in the opposite direction (i.e., exporting traditional functional programming features to object-oriented environments as new design patterns [86]), clearly highlights the relevance of and interest in discussion in this field.

5.3.2 Erlang

Erlang is a functional language that was created as a tool to develop robust applications meant to run over a net of computers. Originated inside Ericsson Telecommunications, its initial aim was to be of use to program tele-

phone switches. However, it turned out to be a programming environment that helped to speed up the development and to reduce the maintenance effort while generating highly reliable robust pieces of software. This was the key for it to step out from the telephony world and to start being used for other purposes. Nowadays, Erlang and the set of libraries that conform the OTP platform (often referred together as Erlang/OTP) have proved that they can be a perfectly valid environment to develop almost any kind of software application, especially when robustness, reliability, high-availability, maintenance ease, and transparent distributability are essential requisites [91–95] (for more detailed information about the language, refer to [96–98]).

5.3.2.1 Object-Orientation and Erlang

Even though Erlang is not an object-oriented language, we can easily program its main concepts in a number of different ways. Several solutions/approaches have been proposed for OO in Erlang over the past few years:

WOOPER This *Wrapper for Object-Oriented Programming in Erlang* project was first released in 2007, as an open-source layer on top of the Erlang language which provides support for object-oriented constructs and features (such as polymorphism and inheritance, for example) [99]. This approach defines each class in a different Erlang module and uses Erlang lightweight processes to represent object instances. It makes use of private hash tables to store object attributes as key-value pairs, and method definition and invocation need to follow some specific conventions.

eXAT The *eXperimental Agent Tool* project integrates a library that allows developers to write Erlang programs with an object-oriented flavour [100]. Again, classes are identified with Erlang modules, and objects are used by means of a special module called `object`, which creates and destructs instances, and also provides access to attributes and methods. The first publications about eXAT date to late 2003 and 2004 [101, 102].

ECT The most recent contribution to this particular research area is *ECT: Erlang Class Transformation* [103], which is to be released soon. Unlike the previous experiences, this approach does not use Erlang processes to map the object abstraction, but instead it extends the language using a series of syntactic sugar definitions that are translated into pure Erlang by a transformation before compilation time. Once more, a class is an Erlang module, but object instances are record instances rather than processes, which allows object pattern-matching.

There has also been partial studies or approximations to some of the most interesting features of OO, such as inheritance [104], but neither these nor either of the previous outlined solutions had been released or even developed

when the ARMISTICE project was started in 2002. Consequently, we developed our own solution for ARMISTICE [105] along the same line that has been fully extended and formalised now in the aforementioned ECT initiative.

For the concept of object class, we chose to map each class to an Erlang module implementing the interface and behaviour of that class. Regarding the implementation of the concept of object state, there are two possible approaches:

- Representing states as explicit data structures. This is simple, but involves passing around such structures within each object message (i.e., function invocation). It also imposes a coarse grain concurrency.
- Representing states as implicit data structures. If each object instance is an Erlang process, then its state is determined by the state of such a process (represented internally by some kind of data structure). This provides secure encapsulation and a fine grain concurrency, more consistent with OO principles.

With the first approach, each class instance is associated with a data structure that represents its members, both state and methods. In this case, if a method call is invoked over an object, a method (function) is evaluated with the actual data structure that models the object as an argument.

To illustrate this, the following code corresponds to the `sit_riesgo` module, which represents a simplified object of interest in ARMISTICE, providing that instances of this class have attributes named *oid* (risk object identifier), *codigo* (code), *nmb* (name), and *grp* (risk group):

```
-module(sit_riesgo).
-export([new/0]).
-export([get_oid/1, get_codigo/1, get_nmb/1, get_grp/1]).
-export([set_codigo/2, set_nmb/2, set_grp/2]).
-record(sit_riesgo_vo, {oid, codigo, nmb, grp}).

%% @doc  Creates a new object
%% @spec new() -> Object :: record()
new() ->
  {ok, #sit_riesgo_vo{}}.

%% @doc  Returns risk object name
%% @spec get_nmb(Object :: record()) -> {ok, string()}
get_nmb(Object) ->
  {ok, Object#sit_riesgo_vo.nmb}.

%% @doc  Modifies risk object name
%% @spec set_nmb(Object :: record(), NewName :: string()) ->
%%                                                      {ok, record()}
set_nmb(Object, NewName) ->
  {ok, Object#sit_riesgo_vo{nmb = NewName}}.
```

```
...
```

Simplicity, low resource usage (one object is a record, or a list of records – for class inheritance), and the fact that objects are really immutable (Erlang data structures are non-destructive) are the main advantages of the use of explicit data structures to implement objects in this functional language. But the simplicity of this approach also has some disadvantages. Using records to implement object breaks the encapsulation principle and forces us to explicitly manipulate object state between messages. This is a very annoying requirement and could be the source of many programming bugs. The next code fragment clearly shows this drawback:

```
...
{ok, RS} = sit_riesgo:new(),
{ok, RS1} = sit_riesgo:set_codigo(RS, "STOR-11"),
{ok, RS2} = sit_riesgo:set_nmb(RS1, "Storage building"),
{ok, RS3} = sit_riesgo:set_grp(RS2, 23445),
...
```

Using the second approach, each class instance is mapped to an Erlang process. The object state is implicitly encapsulated in the Erlang process state. From the client's point of view, an object is now modelled as a process ID (its *object identifier*), and sending a message to an object is, actually, sending a message to the process object. The following example shows our class `sit_riesgo`, had it been implemented using the process approach:

```
-module(sit_riesgo).
-export([new/0]).
-export([get_oid/1, get_codigo/1, get_nmb/1, get_grp/1]).
-export([set_codigo/2, set_nmb/2, set_grp/2]).
-record(sit_riesgo_vo, {oid, codigo, nmb, grp}).

%% @doc  Creates a new object
%% @spec new() -> pid()
new() ->
  spawn(?MODULE, dispatch, [#sit_riesgo_vo{}]).

%% @doc  Dispatches object functionalities invocation
%% @spec dispatch(State :: record()) -> {response, pid(), Result}
dispatch(State) ->
  receive
    {call, Pid, Method, Args} ->
      {ok, NextState, Result} = ?MODULE:Method([State | Args]),
      Pid ! {response, self(), Result},
      dispatch(NextState)
  end.
```

```
%% @doc  Returns risk object name
%% @spec get_nmb(State :: record()) -> {ok, record(), string()}
get_nmb(State) ->
  {ok, State, State#sit_riesgo_vo.nmb}.

%% @doc  Modifies risk object name
%% @spec set_nmb(State :: record(), NewName :: string()) ->
%%                                               {ok, record(), ok}
set_nmb(State, NewName) ->
  {ok, State#sit_riesgo_vo{nmb = NewName}, ok}.

...
```

And it would be used as follows:

```
...
RS = sit_riesgo:new(),
RS ! {call, self(), get_cdg, []}}
receive
  {response, RS, RSCode} -> RSCode
end,
RS ! {call, self(), set_nmb, ["Storage building"]}}
receive
  {response, RS, ok} -> ok
end,
...
```

This second solution is probably a more natural way of mapping many of the object orientation principles: it provides object state encapsulation, object identity is unique (as long as processes identifiers are unique), and these objects are really concurrent objects. However, there is a drawback: this approach is more resource consuming than the former, specially if there are many small objects in the system.

Focusing on our case study, ARMISTICE deals with many *Value Objects* [106]. This kind of object, which maps database objects, is quite simple since its behaviour is mainly constrained to `get` and `set` methods. The simplicity, short life cycle and high number of instances in the system suggest the use of explicit data structures to implement them. On the other hand, sometimes we find it necessary to implement some classes based on the *Singleton* [74] design pattern (the database connection manager, process monitors. . .). This pattern ensures the existence of only one instance of the class at any time and provides a global access point for the whole system. Since these objects are fine-grain and have a long life-cycle, they are implemented using the second solution. Moreover, the global access point is easily implemented using the Erlang process registration facilities [97].

Hence, both alternatives are used for the business objects and logic implementation, carefully choosing the best of the two options in each case. In

general, explicit data structures are preferred when objects have a short lifespan and do not require intense concurrency support, while processes are best suited for long-life objects which need to attend a great number of concurrent invocations.

5.4 Developing key aspects of ARMISTICE

This last section intends to be a demonstration of the arguments that have been previously outlined in this chapter. Namely, we claim that implementing an object-oriented design in a functional language is a feasible task, and that using functional programming as a development paradigm is fully compatible with a regular software development. What is more, it increases the ability of facing complex challenges, enhances the implementation experience, and produces less and higher-level source code that would be inherently easier to debug and to maintain.

As we will see, the use of Erlang and its programming philosophy to deal with the domain complexity of the ARMISTICE project has simplified the development process considerably, reducing implementation time cycles. Moreover, thanks to the specific properties of the Erlang/OTP platform, it allowed to easily provide features such as scalability and high availability at a very low cost.

The most relevant implementation scenarios in which using this technology made a difference are:

- ARMISTICE's formulae and restriction ad-hoc language, developed to help expressing insurance constraints.
- ARMISTICE's descriptive modelling of contractual clauses and policy warranties, including allowance of human-language explanations as integral part of them.
- ARMISTICE's decision support engine, to assist users when selecting the most suitable policy to cover accident damages.

5.4.1 System architecture and technologies

The set of technologies that has been put together to bring this case study project to life is a novelty in the field of traditional management software. The business logic of the ARMISTICE system, as we outlined in the previous chapter, has a high level of complexity, managing different kinds of heterogeneous business objects representing risks, rules modelling the exposures to dangers of such risks, and applicability constraints for insurance policies covering them. This complex domain, in the shape of the application use cases, has been mainly developed using a declarative language: the concurrent functional language Erlang [107].

ARMISTICE has a client/server architecture structured in layers using two well-known architectural patterns: Layers and Model-View-Controller [74]. The *client side* is a lightweight Java [108] stand alone and multi-platform client which just performs remote procedure calls (RPCs) to the server and implements no business logic. We could say that this client only knows how to forward user interaction as specific enquiries to the server, and how to display the answers back on the screen. The absence of responsibility on the user side makes the GUI an easily adaptable or interchangeable component.

Communication between the user side and the server side (which goes through a network) is implemented using XML-RPC [109] as *middleware*. XML-RPC is a remote procedure call protocol [110] that was designed to be as simple as possible, but also flexible enough to transmit complex and customisable data structures.

The *server side*, completely developed in Erlang, supports the model and all the business logic; it is structured in four tiers (see Fig. 5.1):

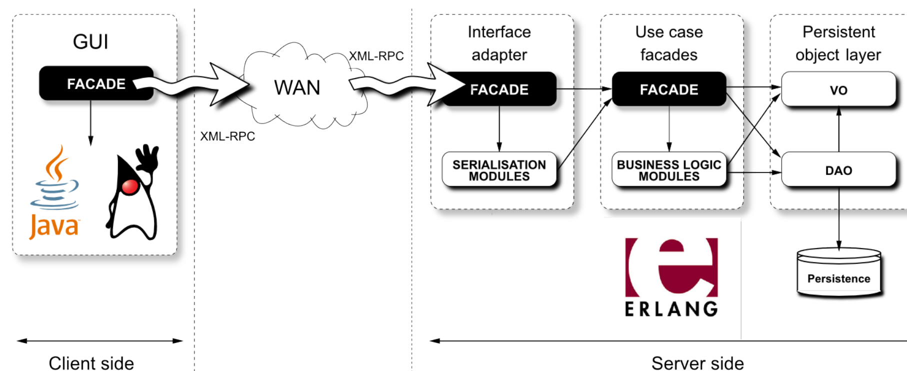


FIGURE 5.1. ARMISTICE architecture overview

- *Interface adapter*: Here is where the communication between client and server occurs. This component receives messages from the clients, deserialises the input and parameters, and gives them the suitable format so that they can be forwarded to the next server software level. One or more operations from the next layer can be invoked, and the output is again given an appropriate format before being sent back to the requesting client.
- *Use case facades*: These are the access points to the business logic, each of them representing a set of related application use cases. Each facade refers to the methods implementing the use cases, and may also use the persistent objects layer.
- *Persistent object layer*: In this tier, business objects are defined using the Value Object (VO) and Data Access Object (DAO) patterns [106]. Each

object models a domain entity or business concept, and this layer takes care of their permanent storage interacting with the persistence layer.

- *Persistence*: This last component represents the permanent storage in a relational database.

The remainder of this chapter focuses on the implementation mechanisms and solution details related to one of the most relevant ARMISTICE use cases: determination of the coverage for an accident. Starting with the formulae and restriction language that was developed to help expressing coverage constraints, then proceeding with policies modelling, and finally describing the decision support system built on top of them.

5.4.2 Formulae and restriction language

ARMISTICE's formulae and restrictions, previously described in Chapter 4 (from page 48 on), are built using an *ad-hoc* language that was developed to allow all the expressiveness and flexibility that was required to properly and powerfully model the risk management business domain.

Our restriction language is actually a superset of the formulae language, where logical operators and the concept of *nuance* ($s \in String$) have been added. The possibility of including these human-language explanations as part of the description of a supplement clause and the ability to deal with them at the same level as other parts of the constraints expressions allows ARMISTICE to very faithfully represent reality, and makes it possible for its decision support system to assist the user to obtain extremely accurate results when selecting the appropriate policy to charge with the expenses of an accident.

As explained when the formalisation of ARMISTICE's business domain was described (viz, section 4.5.3), the key objects to implement such decision support task are *Conditionals*. A conditional is an insurance policy element that is thoroughly analysed when looking for coverage to any claim, since it is the model of the contractual clauses of the policy. In other words, it is the model of the policy coverage, the model of the policy *warranties*, the model of the policy terms (i.e., applicability preconditions, excess, and limit):

$$\mathcal{C}_i = \{g_0, \dots g_n\} \quad / \quad \forall x \in [0, n],\ g_x = (Restriction, Excess, Limit) \quad (5.1)$$

[1] Actually, $Excess, Limit : \mathcal{F}or \to Money$.

where $Restriction \in \mathcal{R}es$ and $Excess, Limit \in \mathcal{F}or$[1] are the corresponding coverage conditions, excess and limit. The logic restrictions $\mathcal{R}es$ and numeric calculations $\mathcal{F}or$ were defined as:

$$
\begin{aligned}
\mathcal{R}es_i \quad &: \quad \{f_j \;/\; f_j \in \mathcal{F}or\} \\
&\times \quad \{h_k \;/\; h_k \in \mathcal{H}\} \\
&\times \quad \{s_m \;/\; s_m \in String\} \quad \longrightarrow \quad \{Boolean \cup String\} \times Integer \\
\mathcal{F}or_i \;&:\; \{p_j \;/\; p_j = (a, v) \wedge a \in \mathcal{A} \wedge v \in Type(a)\} \longrightarrow \cup\, Types
\end{aligned} \tag{5.2}
$$

where the p_j operands are either user-defined values or risk situations properties, that is

$$
\mathcal{F}or_i : \{\mathcal{RS}_j^{v_x, r_y}.p \;/\; \mathcal{RS}_j^{v_x, r_y} \in \mathcal{RS}\} \times p_{user} \times p_{sys} \longrightarrow \cup\, Types \tag{5.3}
$$

where p, p_{user} and p_{sys} are sets of pairs label/value $(a, v), a \in \mathcal{A} \wedge v \in Type(a)$ which can be accessed using specific constructors of the modelling language. The p_{user} set, also known as *user parameter* set, are input values provided by the user in the process of evaluation of a formula. The p_{sys} set, also known as *system parameters*, are system-scope values, whether they be calculated when the formula is evaluated or through time as internal counters.

The system provides a collection of tools designed to manage the elements in the $\mathcal{F}or$ and $\mathcal{R}es$ sets. Restrictions are the conditions to be held by a warranty in order to be activated, and every warranty (i.e., contractual clause) inside a conditional of a policy has an applicability restriction, which models its behaviour. These expressions can access the context where they are evaluated and they can use temporal information about risk situations, policies, and other claims. The set of constraints and calculations, together with the risk situations meta-description (i.e., risk groups) and the set of hazards, are the essential elements to build a flexible and adaptable RMIS framework.

5.4.2.1 Formulae

Formulae, together with the meta-description of insurable elements and the set of hazards, are an essential element as far as ARMISTICE's adaptability and extensibility is concerned. The specific language, designed and developed to express formulae, includes basic operators, grouping operators, and conditional operators, as well as constructs to access risk situation properties, policies internal data, system counters, user input parameters, etc. One can also use already defined formulae as part of a new formula, enabling the reuse of formulae definitions and the creation of formulae libraries. Hence, there is great flexibility when it comes to modelling receipts, excesses, and limits calculations.

As a sample of this language, the following text is a possible way of modelling a formula to express the excess of a warranty, just as the ARMISTICE user would write it down in a GUI text field. Consider an insurance excess which compensation is equivalent to the total cost of the damages, for losses under a minimum value, otherwise being equal to a parametrisable percentage of the total cost damages. Consider also the existence of an upper limit equal to the average cost of storage valuables in the risk situations that have been affected. Such a formula would look like:

```
if ( accident.covered < #min,
     accident.covered,
     min(#percentage * accident.covered,
         average(policy.supplement.risksituations(%stock))) )
```

In the previous example, different kinds of parameters and operators are used: two user input parameters (`#min` and `#percentage`, that for each particular context in which this formula is used could take different values), a grouping operator (`average`), a global counter (`accident.covered`), and an access operator to a property from a set of risk situations (`%stock`) covered by a policy (`policy.supplement.risksituations`) supplement.

5.4.2.2 Restrictions

A conditional constraint clause (i.e., restriction) is to be evaluated to a boolean value, $true$ or $false$, reflecting whether the associated warranty can be applied to a particular claim or not. However, sometimes it is not possible to state directly the restriction result. In these situations, the truth value of a restriction may depend on the answer of the user to a question expressed in natural language (namely a combination of nuances, $s_m \in String$), therefore, it can only be evaluated to a boolean value by means of human intervention.

The evaluation process of a restriction can be broken down into the evaluation of each and every one of its terms, and then their combination into a global result, either a truth value, or a simplified expression with a logical combination of nuances that have to be decided by the user. Besides, every evaluated restriction has an associated weight expressed as an $Integer$ value (see equation 5.2) which heuristically measures the relationship between the context information, the present hazards, the affected risk situations, and the restriction itself. This weight can also be used as a sorting criterion among the automatically undecidable contractual clauses.

A very simple example of a restriction, containing two elements of the hazards set $\mathcal{H}$ and two nuances, is displayed next:

```
        (fire and "Not arson")
    or
        (earthquake and "Great magnitude")
```

where `fire, earthquake` $\in \mathcal{H}$. In a context in which the danger causing the accident is a fire, the previous expression will be simplified for the user by posing the question "*Arson*?"

5.4.3 Policy relevance

As has already been stated, ARMISTICE's business logic has been implemented in Erlang using a combination of object-oriented and functional programming styles. Object-orientation has helped modelling the business domain, identifying business concepts, and representing them as objects with certain properties and functionalities. For the actual implementation of those functionalities, whose complexity is directly proportional to that of the business domain itself, a conventional functional style has been put in practise.

The use of the functional abstractions that have been described earlier in this chapter has been a key factor in the simplification of the development process. To illustrate this idea, an outline of one of the key business processes of the system is shown next: the assistance in the process of determining whether a certain insurance policy is suitable to provide coverage for a given loss or accident.

Applicable policies are those whose warranties and constraints allow them to be charged with the expenses of facing the risk that has caused damage. In order to decide which policies, among all the policies managed by ARMISTICE, are relevant, insurance policy clauses (warranties) have been modelled as restrictions using an ad-hoc restriction language defined and implemented using Erlang data structures.

A simplified implementation of such modelling, expressed in an Erlang-like syntax, is:

```
    Constraint =   {literal,  Nuance}
                 | {danger,   Danger}
                 | {negation, Constraint}
                 | {all,  [Constraint]}
                 | {any,  [Constraint]}
```

When looking for a suitable policy to cover the damages of a given accident claim, each and every of its insurance clauses is matched against the specific details of the hazard causing the accident. Here, the use of pattern matching is of enormous help, considerably simplifying the implementation of this process and, at the same time, descriptively documenting it.

Following the simplified example before, the accident information would be reduced to the main causing hazard:

```
% @doc Determine response of a constraint evaluation
% @spec relevant(H :: Hazard(), Clause :: Constraint()) ->
%                                             boolean() | string()
relevant(H, {literal, Nuance}) -> Nuance;
relevant(H, {danger,  H})      -> true;
relevant(H, {danger, NH})      -> false;
relevant(H, {negation, C})     -> do_not(relevant(H, C));
relevant(H, {any, Clauses}) ->
  lists:foldl(fun(A, B) -> do_or(A, B) end,
              false,
              [ relevant(H, C) || C <- Clauses ]);
relevant(H, {all, Clauses}) ->
  lists:foldl(fun(A, B) -> do_and(A, B) end,
              true,
              [ relevant(H, C) || C <- Clauses ]);

% @doc Perform NOT operation on a value, which can be either a
%      boolean value or a string containing some nuances
% @spec do_not(Op :: boolean() | string()) -> boolean() | string()
do_not(Op) when atom(Op) -> not Op;
do_not(Op) when list(Op) -> "NOT" ++ Op.

% @doc Perform OR operation on a pair of values, which can
%      be either boolean values or strings containing nuances
% @spec do_or(OpA :: boolean() | string(),
%             OpB :: boolean() | string()) -> boolean() | string()
do_or(OpA, OpB)   when atom(OpA), atom(OpB) -> OpA or OpB;
do_or(true, OpB)  when list(OpB)            -> true;
do_or(false, OpB) when list(OPB)            -> OpB;
do_or(OpA, OpB)   when list(OpA), list(OpB) -> OpA ++"OR"++ OpB;
do_or(OpA, OpB)                             -> do_or(OpB, OpA).

% @doc Perform AND operation on a pair of values, which can
%      be either boolean values or strings containing nuances
% @spec do_and(OpA :: boolean() | string(),
%              OpB :: boolean() | string()) -> boolean() | string()
do_and(OpA, OpB)   when atom(OpA), atom(OpB) -> OpA and OpB;
do_and(true, OpB)  when list(OpB)            -> OpB;
do_and(false, OpB) when list(OPB)            -> false;
do_and(OpA, OpB)   when list(OpA), list(OpB) -> OpA ++"AND"++ OpB;
do_and(OpA, OpB)                             -> do_and(OpB, OpA).
```

Representing policy clauses as lists of restrictions, the complex selection of the relevant clauses is easily written down as a filter, using a list comprehension:

```
% @doc Determine if an insurance policy is applicable
%      to a certain accident claim
```

```
% @spec applicable(Accident :: Claim(),
%                   Insurance :: Policy()) -> boolean() | string()
relevant_policy({accident, Hazard}, {policy, Clauses}) ->
  [ ClauseID || {ClauseID, ClauseDefinition <- Clauses,
                relevant(Hazard, ClauseDefinition) =/= false ].
```

It is clear that working at the higher abstraction level that functional languages provide simplifies enormously the implementation of complex behaviour without giving up any expressiveness, flexibility, or power.

5.4.4 ARMISTICE as decision support system

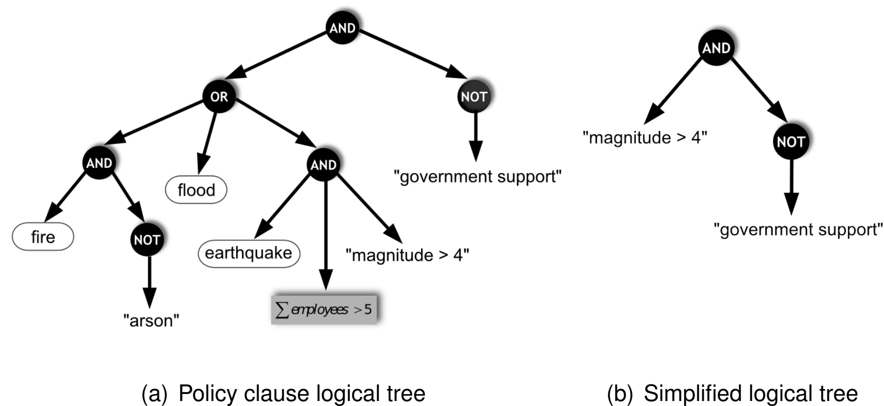
Thanks to the detailed design exposed in Chapter 4, which paid a lot of attention to all domain properties and characteristics, ARMISTICE is a very powerful tool for the daily control and management of accident claims. It improves the risk management decision process at two different levels: as a proactive working tool for the non-expert user, and as an advanced analysis tool for the expert user.

Whenever a potential risk (or several) turns into a real accident that actually affects one or more of the risk situations handled by the system, the question arises of whether there may be one (or more than one) insurance policies whose coverage terms include the damaged objects of interest, to be charged with the recovery expenses (and, if several, which would be the best to demand the repayments from, for that particular accident). To answer all these questions and manage the loss situation, a file is opened by the insurance department to manage the corresponding claim.

Storing all the information about the contracted policies, ARMISTICE is capable to act as a decision support system, discarding all the irrelevant policies (those with non-applicable warranty clauses, covering different risk situations, different hazards, or different time periods) for a given accident. It does so by automatically checking all policy data (specifically, each supplement/conditional data) against the known accident details the user inputs. By analysing policy warranties contents (i.e., evaluating the associated restrictions) and contrasting them with accident dates, objects of interest involved, materialised risks, etc., all non-applicable warranty clauses (and thus, all non-applicable supplements, then policies) can be discarded, leaving for the user to select from only a few choices, corresponding to those constraints whose applicability lays on the human-language nuances they contain, hence only decidable by a human being.

This process can be seen as the simplification of a logical tree representing the contractual clauses of a policy. The way it is performed consists in pruning branches off that tree, using context information. Prunable constraints will be those that can be fully evaluated and so automatically designated as *true* or

false by the system. Let us picture, for instance, an applicability precondition of a hypothetical contractual clause that would provide coverage against fire (but only if it is not arson), flood, and earthquake (but only if its Richter magnitude is greater than 4.0, and whenever the total number of employees in the set of affected risk situations –i.e., company's offices– is greater than five). Besides, let us say that coverage would also only be supplied if the government does not provide financial support to alleviate the accident. These restrictions can be logically organised as shown in Figure 5.2(a).



(a) Policy clause logical tree (b) Simplified logical tree

FIGURE 5.2. Policy clauses representation

Now, if an earthquake strikes the region and there are more than five employees working in the affected facilities, the system can automatically simplify the expression to Figure 5.2(b). The user that inputs the original information about the earthquake and its effects into the system, should now just answer whether there will be government response to the catastrophe and if the tremor had a relevant magnitude for the clause to be applicable (i.e., *true*).

So the output of the decision support module is, in the end, the list of policies which are either downright candidates, or else that have non-automatically decidable constraints/clauses, expected to be a much more smaller set than the original one. These fewer possibilities can then be explored by the non-expert user, to decide if the nuances are important or not, concerning the specific accident that she is dealing with at the moment, finally picking the most convenient coverage choice.

Having the system discarding all non-relevant policies, and reporting back just those either already applicable, or which require the user to make a decision on about their suitability because they include human-language nuances in the warranty clauses text, turns the decision making process into something much easier, even for the non-expert user, since the quantity of information to be taken into account is substantially reduced. Besides, from the Artificial Intelligence perspective, a correspondence can be established among the different parts of a classic production system (cf. Fig. 5.3) and some of the

ARMISTICE components we have just described. Such a production system would consist of:

- A *knowledge base*, containing
  - *a rule base* (or list of rules), represented by the applicability constraints of the insurance warranties of each policy;
  - *a factual-data base*, containing all the information about accidents that have happened, as well as definition and properties of risk situations and policies, and their evolution over time.

  The combination of the facts on the factual-data base and the rules on the rule base is the trigger for the relevant rules activation (i.e., selection of accident-relevant policies), according to the actual characteristics of what has indeed happened.

- An *active memory* storing the results that the system is producing, as well as the user input about the facts, relevant active rules, etc. This component will, hence, have information about the set of policies that are retrieved after matching the warranties constraint definition against the data about an accident. As far as ARMISTICE is concerned, a rule/warranty that is fired/activated means the policy it belongs to is potentially appropriate to cover the expenses the accident recovery is going to cause.

- Last but not least, an *inference engine*, realised in the restriction and formulae analysers which conduct the whole process, finally suggesting the coverage after calculating conditions, excesses, and limits. Its main properties are forward chaining, activation of all matching rules, and exhaustive depth first search (meaning that all sub-constraints and sub-formulae are analysed in-depth).

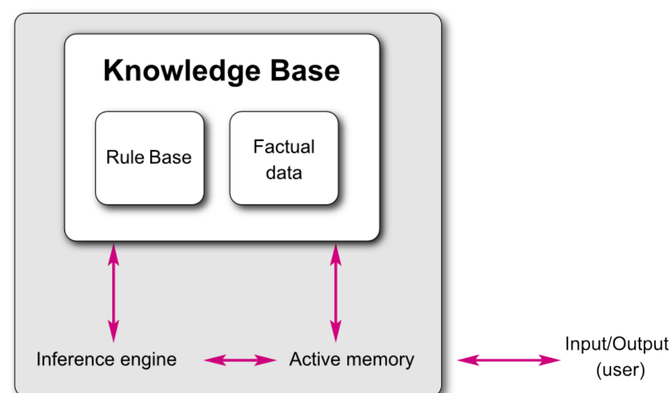


FIGURE 5.3. Architecture of an AI production system

Once these initial steps in the accident registration have been performed, the usual activities involved in accident claim management procedures take place: determination of the tasks to be carried out to repair the damages, sending and receiving evidence documentation and various paperwork, issuing and processing payments, claiming and recovering indemnities, etc.

As repairs are being performed, receipts are coming in, and insurer compensations are being received, the corresponding data helps the system keep the claim status up-to-date, right through and up to the final stage when everything is solved and the file is permanently closed. Even then, ARMISTICE's usefulness is not finished. Apart from these everyday kinds of operations, there is potential for analysis that can be performed on the basis of all the daily information gathered. This is the second risk management working area that ARMISTICE greatly improves upon.

The head of the risk management department will have the actual data her company is producing added to her regular statistical resources, reports, and studies, right away. At any time, she can query the system and obtain different flavours of reports that will show if the risk management policy is doing its job, if the losses are being recovered as desired, if any of the contracted policies are redundant or superfluous, if there is any hazard causing uncovered accidents because it was missed or underestimated at insurance negotiation time. This second task is even more critical than the first, because it can help to detect deviations in the risk management policy at relatively early stages and, hence, to correct them.

Such in-depth analysis is only possible thanks to the application managing all relevant information, from risk situations and hazards to policies and accidents, a direct consequence of the effort put in the analysis and design stages, and made possible by the high-level of abstraction kept at the implementation, too. There is no doubt that this has made ARMISTICE a very powerful resource for those responsible to have the overall risk management policy of a company under much higher control.

6

Ensuring functionality and quality through testing

As part of the natural evolution of the discipline, software engineering is nowadays undergoing a progressive displacement of concerns, that are moving from functionality-related aspects towards quality assurance efforts. Functionality and quality are, however, two sides of the same coin, and the common path of software validation pursues both.

As software becomes more and more ubiquitous, the complexity of the problems that need to be addressed increases as well. Greater challenges require not only better analysis, design, and implementation capabilities, but also more effective techniques when it comes to providing some measurement of final product quality and expectations compliance. This is why software companies pay more attention now to their engineering processes [111], including (and especially) their software testing stages, as part of their quality improvement policies. The objective is, of course, to enhance the properties of the delivered software, in particular by reducing the amount of faults that make their way into the deployment and production stages.

Crucial as it is acknowledged to be to software engineering, software testing still does not offer a degree of discipline and results comparable to other software engineering activities. Testing goes beyond debugging: it means evaluating behaviour, attributes, or capabilities of programs or systems to determine if they meet their requirements. And even though our way of implementing activities, modelling, and representing properties or services has evolved enor-

mously, we still use mostly the same testing procedures invented decades ago. They involve, in many cases, creativity; they require, to be more efficient, certain experience; they benefit from intuition. We approach much more complex business domains and industrial and social needs, but fault identification and correction processes have not stepped up to the same level as the software under test. Last but not least, testing is usually a trade-off between project budget, development time, and product quality. And while software testing may be expensive, not testing it is even more costly [112].

With the complexity barrier getting pushed forward day after day, the furthest we can go is constrained by the strength of the techniques we put in practise in each step of the development process. This is why, in this dissertation, we are focusing on every stage of software building, seeking how to improve software design and construction as a whole. In this chapter, we aim to show that the benefits of the paradigm we have been discussing extend to testing, enabling the application of certain methodologies which produce excellent results with moderate effort. We will see that, thanks to the use of functional programming, this essential task is greatly improved in many ways, such as easiness, rigorousness, and effectiveness.

6.1 Software testing

The distinction between debugging and testing was initially introduced by Glenford J. Myers in 1979 [113], as he wanted to express the need of the software engineering community to regard verification as something else. Some years later, in 1988, Gelperin and Hetzel [114] established a taxonomy of the phases and goals through which software testing had evolved:

First phase (before 1956): debugging

In the early days of software development, testing efforts were mostly reduced to achieving program execution. Most of the testing efforts were focused on hardware, so terms such as program checkout, bug inspection, fault correction, and even testing were not clearly differentiated.

Second phase (1957-1978): demonstration

Around the fifties, the general view of testing started to change and activities aimed at fault detection were clearly distinguished from fault location, identification, and correction tasks. The objective was to provide some empirical data about the software behaving as expected according to its specifications and requirements.

Third phase (1979-1982): destruction

Criticisms of the demonstration-oriented testing appeared, arguing that the risk of introducing unintended or subconscious trends when selecting data to test for successful execution could lead to poor testing results. As a consequence, a new perspective is proposed, and testing

is, as a result, oriented to detect implementation faults by trying to force software crashes.

Fourth phase (1983-1987): evaluation
As software professionals gave in to the fact that no testing process could guarantee error-free software, the belief spread that at least a carefully chosen set of techniques could help ensure development of better quality software. Hence, testing enters each life-cycle phase, which has now a set of activities or sub-products to be verified, in an attempt to detect not only implementation faults, but also requirements and design defects.

Fifth phase (from 1988): prevention
The last phase represents the generalisation of the evaluation-oriented testing to all levels of testing, seeking rather to prevent than only to detect requirements, design, and implementation problems. Timely test planning, test analysis, and test design forces development teams to reflect about verification and validation activities as early as requisite gathering starts, potentially revealing flaws, incompleteness, ambiguity, inconsistencies, and/or incorrectness before they are carried into the following development stage.

The importance of testing activities has, thus, grown in parallel with its relevance within software development, from merely test execution and closing stages of a project to a key task that interacts with all development steps throughout all project lifespan. In this thesis, we aim to contribute to the consolidation of testing as a first-order activity, which is planned and thought about already from the analysis and design activities. Indeed, our case study formalisation in Chapter 4 will be helpful again, as we will see in Section 6.3.3.

6.1.1 Verification versus validation

The terms *verification* and *validation* are commonly used interchangeably in the industry, and it is not uncommon to see them incorrectly defined. The Capability Maturity Model Integration (CMMI) defines these concepts according to the IEEE Standard Glossary of Software Engineering Terminology:

Verification “Process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.”

Validation “Process of evaluating a software system or component during or at the end of the development process to determine whether it satisfies the specified requirements.”

In other words, verification activities try to give an answer to the questions “*are we building the product right, does it conform to design specifications*?”, while

validation tasks attempt to shed some light over the complementary questions "*are we building the right product, does the product do what the user requires*?"

Different testing strategies face software verification and validation from a diversity of perspectives, seeking from empirical information to formal evidence about the quality of a product or service. The stage or stages of the development process they are used in also vary (as do their methods and tools, and its formality and rigour), although most test efforts are employed after the requirements have been defined and the coding process has been completed at least to a certain extent. As different as they may be, the main concern of all testing alternatives is common: to be as effective as possible in minimising efforts and (side) effects.

In addition, one should consider that software testing effectiveness is a parameter of the software itself, strongly dependent on its properties and the characteristics of its target audience. Take, for instance, a flight simulator game and a flight simulator instructor for pilots. Being products with similar features, the error tolerance is completely different in these two scenarios; in other words, the same kind of software failure will not be equally acceptable for each of them. Based on this last criteria, certain testing methodologies may be better suited than others, or even be not applicable at all.

And even when using the most appropriate testing approaches, we should bear in mind that the search for software errors is a non-decidable search problem [115] with an infinite space:

- It is a search problem because the space of all possible test scenarios is explored to find the ones leading to bugs.
- It is a non-decidable problem because the decision whether a certain behaviour is indeed defective is not inherent to the testing process, it usually requires external input (i.e., the developer, the user).
- It is an infinite-space search problem because the total number of possible scenarios is often extremely large, practically infinite.

In this dissertation, we use the term 'testing' as a general concept, and refer specifically to 'verification' and 'validation', according to the previously mentioned IEEE definitions, whenever we want to specifically emphasise one of the two perspectives.

6.1.2 Testing levels

A common categorisation of testing [116] approaches corresponds to the well-known levels of testing (as first defined in the V model, cf. page 13), that includes:

Unit testing Also called *module testing* or *component testing*, unit tests are typically written and run by software developers themselves, and consists in choosing the smallest testable part of an application or system (i.e., a function, a module, a program) and checking whether it meets its design and behaves as intended. As many other testing activities, unit testing produces a rudimentary form of system documentation, since the tests themselves represent a form of written contract for a piece of code that, among other things, allows to combine source refactoring and retesting to ensure it still works as expected (i.e., *regression testing*, cf. page 85). Even though it will obviously not detect complex errors (such as integration errors, broader system-level errors), a large percentage of defects are identified thanks to this testing modality.

Integration testing Modern systems are usually structured in separate modules that work together to offer full services or provide complex functionalities. Apart from being unit-tested individually, these software modules are combined and tested as a group in integration testing activities. The assembly needs to interact correctly across procedure calls or process activations, to verify not only functional, but also performance and reliability global requirements. Thus, this testing technique exercises components interfaces in a *black-box testing* style (cf. section 6.1.3.2). Following a bottom-up approach, the testing groups grow gradually, eventually leading to the next testing granularity.

System testing Using the functional requirements and/or the system requirements specification, a complete, integrated system can be tested as a whole to evaluate its compliance with those requirements. This system testing is usually again a *black-box testing* scenario, which creates artificial environment conditions to lead the system to different situations using no information of the system internals. This modality tests not only design and behaviour, but also a whole set of different aspects which become relevant at this level: performance, security, scalability, capacity, or recovery.

Installation testing In many cases, the installation of a software application is the first interaction with real users, which should make the verification of this process a main concern. Instead, it is often one of the most under-tested aspects. Actions devoted to ensure that every possible configuration receives an appropriate level of attention, to grant that all installed features and options function properly, and to verify that all needed components are indeed installed, is called installation testing. It seeks to increase the level of confidence with which the software is delivered to clients. This is especially important whenever the software is to be released into already 'live' target environments, where unexpected events could cause data loss or corruption.

Acceptance testing Though it can also be performed by the software provider, the leading actor of acceptance testing (also called *functional testing* or *validation testing*), the final stage of testing before product release or deployment, is generally the end-user. The objective of acceptance tests is to provide confidence in the delivered software meeting the business requirements, on the basis that if a system works as intended and without issues during 'normal use' testing sessions, the same level of stability in production can be inferred. To accomplish this, the testing environment needs to be as close as possible (if not identical) to the anticipated deployment environment, and a trial-and-review process needs to take place in the hands of subject matter experts.

Acceptance criteria highly depend on the nature and goal of the software, but they always should be established in advance, derived from the same user stories the system requirements are elicited from. Since, one hopes, previous testing activities will have uncovered operational flaws, this *beta-testing* activity is the final quality gateway.

The contributions in this chapter fall, as we will see, in the first two sets of the previous taxonomy: unit testing and integration testing.

6.1.3 Testing techniques

Next, a short description of the most important testing techniques [117–120] is exposed.

6.1.3.1 Dynamic and static verification

Dynamic verification is the kind of testing that is usually associated with the plain 'testing' concept itself. All the techniques and procedures that are grouped under this umbrella-term require the execution of the software under consideration, and propose a form of review process of its behaviour.

In contrast with dynamic verification, **static verification** (or *static analysis*) does not require the execution of the program or component, and replaces it by its physical inspection. Among the most well-known static verification techniques, we find:

Software inspection Software review, and more specifically, software peer review, is a peer review process in which software professionals different from the software authors go over an application's or component's source code to technically evaluate it for content and quality. The primary objective of software inspection is, according to the Capability Maturity Model, "detecting and correcting defects in software artifacts, and preventing their leakage into field operations". To be effective, software inspection is formally engineered as any other software development

procedure, including planning, preparation, and actual inspection activities, even in an iterative fashion.

As part or consequence of software review, bad practises are detected and source code sees its structure, readability, and maintainability improved, acknowledging verification of source code conventions and application of refactoring actions.

Software metrics calculation Classical engineerings tend to regard measurement as one of the key aspects of management and evaluation. For years, computer science has tried to apply a similar approach to software development. However, software is a much more abstract construct than bridges or machinery. Still, some meaningful aspects can be measured while developing a system, such as performance or code coverage. Other metrics have been revealed as naïve and simplistic metrics, like lines of code, function points, or cyclomatic complexity. Consequently, they have been replaced by maintenance-time ones, such as faults-slip-through [121], aimed to provide feedback for quality evaluation.

Formal verification Instead of trial-and-error, formal verification (also referred to as *formal methods*) tries to prove the correctness of certain behaviour (algorithm) with respect to a formal specification (property). To do so, it pursues the construction of a formal proof on an abstract mathematical model of the system. Such a mathematical model can be a finite state machine, Petri net, labelled transition system, process algebra, operational semantics, denotational semantics, Hoare logic, etc.

There are two broad approaches to formal verification:

- *Theorem proving* (also known as *logical inference*) which, in a strong parallelism with classical logic, consists of the development of formal proofs that show that some statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses). This can be used in a wide variety of domains, given an appropriate formulation of the problem as axioms, hypotheses, and a conjecture. One of the fields where theorem proving is applied is software testing, where the conjecture (model) is that a specific piece of source code (or an entire system) carries out a certain task or provides a certain functionality, according to its specification (formal proof).
- *Model checking*, given a model of a system, seeks to automatically test whether it meets the corresponding specification by means of a systematic and exhaustive exploration of all model states and transitions. That is, once a correspondence between the software under test and a suitable abstraction is established, the problem

is stated as the reachability of a property from an initial state, provided a certain program structure.

Formal verification is used by many leading hardware companies, but its use in the software industry is still languishing. The reason for this is probably the greater commercial significance that hardware errors have, together with the infeasibility of exercising a realistic set of possibilities by simulation, and the suitability of hardware-specific challenges (simpler, with fewer possible outcomes) for automated proof methods, all these making formal verification more productive, and hence, easier to introduce.

The testing techniques we will develop and present later on in this chapter are dynamic verification strategies.

6.1.3.2 White-box and black-box testing

An alternative (and non-exclusive) classification for testing methods is based on the knowledge about the system that is used or needed to perform testing activities. According to this criteria, **white-box testing**, also known as **glass-box**, **logic-driven** or **design-based testing**, derives test cases from the physical program structure.

Examples of white-box testing strategies are:

Code coverage It is one of the first techniques that were designed for systematic testing. It is a form of unit testing that measures to which extent the source code of a program has been tested. According to granularity of coverage, code inspection is sometimes sub-divided into:

- *Statement coverage*, which ensures each line of code has been executed during testing.
- *Decision coverage* or *branch coverage*, which takes care that each control structure that implies a logical decision has been taken in all its alternatives.
- *Function coverage*, which controls that every function has been called.
- *Entry/exit coverage*, which checks that every possible call and return pair has been executed.
- *Path coverage*, which aims to ensure that every possible route through a piece of code, or all independent paths within a module, have been tried at least once.

These levels of code inspection are closely inter-related and they even imply each other in some cases, such as path coverage, which implies decision, statement, and entry/exit inspection; or decision coverage, which implies statement inspection. Besides, other considerations such

as execution of loop structures at their boundaries or checking internal data structures to ensure their validity are often taken into account as well. All in all, full path coverage is often impractical or even infeasible, not to mention undecidable.

Mutation testing Mutation testing consists in modifying source code portions in small ways (either at compile time or at runtime) to mimic typical programming errors and see how tolerant the system is to them, or else to locate weaknesses in sections of the code that are seldom accessed during regular execution. This kind of testing is intended to test the quality of previous testing efforts, since after mutating source code statements, previous test suites will be run again to check whether or not they are able to find those errors. Mutation testing, thus, is an attempt to face the problem of test suite accuracy measurement, using figures such as the ratio of detected mutants to the total mutants created (i.e., sensitivity). Downsides include the strong dependence between the types of faults the mutation operators are designed to represent and its effectiveness, as well as how expensive it is to perform, especially when dealing with large applications.

Fault injection Apart from faulty code, software liability is intimately related to erroneous input data. While consequences of internal failures are likely to be revealed by many different testing approaches, only fault injection can quantify the influence of external events on system behaviour. This testing method consists, then, in using specially crafted input data in an attempt to cause on purpose the software to crash or behave incorrectly, so that error-handling code paths are exercised. The intention is to be able to observe system responses whenever a fault occurs, revealing (under controlled anomalous circumstances) how 'badly' the software can behave. Fault injection can help to determine whether or not a system is able to produce acceptable results in the presence of corrupted or malicious input (i.e., tolerance).

As with mutation testing, the biggest issue here is the plausibility of the injected anomalies which, together with the absence of help tools, makes this strategy rarely feasible.

Besides, all static testing techniques are white-testing by definition, too.

Black-box testing, also called **data-driven**, **input/output-driven** or **specification-based testing**, takes an external perspective. Applicable to all granularity levels, test cases are pairs of valid or invalid inputs and corresponding correct outputs. It is argued that black-box testing represents a more objective approach compared to white-box testing, since it does not need any knowledge about the internals of the software. On the other hand, not using a blind-exploration approach, white-box testing is often more efficient.

Some examples of black-box testing are:

All-pairs This is an effective test case generation technique that avoids test generation combinatorial explosion. Based on that most faults are caused by interactions of at most two factors, pairwise testing tests all possible discrete combinations of pairs of input parameters to a system. This is of course faster than an exhaustive search of all combinations of all parameters, even more if combined with parallel execution of the tests. However, test combinations of pairs may still be infeasible.

It is also referred to as *pairwise testing*.

Boundary value analysis Experience shows that boundaries of input and output ranges of software components are common locations for errors. This kind of testing is actually a test case design strategy which concentrates testing effort on cases near the limits of valid ranges (which are considered error-prone areas), forcing the exercise of the extreme conditions in order to detect problems. Boundary value analysis is often considered a part of stress and negative testing.

Decision table testing Decision tables are used in logic to define rules based on the matching of different actions to a series of conditions. Using a decision table to depict relationships between input data and behaviour or output data, it becomes an instrument to build test cases. Apart from documenting a system by explicitly associating conditions with actions to be performed, and matching many independent conditions with several actions, the effort which is put on the recognition and development of these decision tables may reveal software redundancies and/or inconsistencies.

Equivalence partitioning This technique, also intended for reducing the total number of test cases to be developed, is based on classifying the input data of a software unit into several disjoint partitions and designing test cases to cover each partition once. This approach is based on the assertion that selection of just one test case out of each partition is enough (since using more or even all test cases for the same partition will not result in finding more faults), and also the premise that each partition member causes the same kind of processing and output. However, to be sure about this, a *grey*ish *box testing* may be needed (i.e., considering knowledge about internal data structures and algorithms only for test case design purposes). The classification of input data into partitions that act as equivalence classes is done by referring to input conditions: for each input condition, two equivalence classes are defined, the cases which satisfy the condition and the cases which do not. Boundary value analysis can be seen as a kind of equivalence partitioning.

Exploratory testing Although regarded by many as just “unorganised manual testing”, exploratory testing (or *ad-hoc testing*) is considered by many

others as a test approach that is applicable to any testing technique, and whose main principle is the emphasis on the personal freedom and responsibility of the professional tester. By simultaneously learning, creative test designing, and test running, the tester gathers knowledge about the software that, together with her experience and intuition, leads her to generating new good test cases. The more the tester knows about the system under test and different test methods, the better the testing will be, since the key to test optimisation is no other than the tester cognitive compromise.

Exploratory testing can be one of the more agile and cost-effective testing approaches, since it requires less preparation while revealing important bugs faster. It is also more intellectually appealing, as the tester does not need to complete a series of scripted tests before focusing or moving on to new aspects. Tradeoffs include the impossibility of reviewing tests in advance, and also the difficulty to reproduce them later on.

Fuzz testing Fuzz testing is a simple technique which consists of providing invalid, unexpected random data (also known as *fuzz*) as system input to force failure occurrence and detection. Rather than attempting to guess what data is likely to crash an application (as fault injection would do), fuzz testing simply feeds the software with indiscriminate input. As elementary as it is, this can be a powerful, efficient, and cost-effective approach, and reveal important bugs, often the kind of defects that would be overlooked when software is written and debugged. For instance, fuzzes work best for problems such as buffer overflows, cross-sit scripting, denial of service attacks, format bugs, or SQL injections.

Model-based testing In model-based testing, test cases are (often automatically) derived completely or partly from a model that describes all or some aspects of the system (usually functional properties). Such a model is generally a partial, abstract representation of the system behaviour. Even though the functional model which derives the test cases can be based on the existing source code, model-based testing is still seen as black-box testing variant. The interest in model-based testing is due to its potential for automation: if the model is machine-readable and has a well-defined behavioural interpretation (for instance, can be interpreted as a finite state machine), test cases can be (more cost-efficiently) automatically generated. This promises to increase effectiveness and shorten the testing cycle.

Regression testing Whenever modifications are made to existing software, either during its development or maintenance, experience says that there is a high probability of fault re-emergence. This is typically due to the fragility of bug fixes, which instead of solving the problem, patch it for a narrow set of scenarios directly related to the observed anomaly, but not for the general case. This kind of testing aims to reveal such software

'regressions', functionalities that were once correct but stop working as expected as unintended consequence of program changes.

Common regression testing methods include re-running all previously designed tests, which is the reason why most test suites are built in an incremental way. However, for many projects it is not feasible to follow an approach like this, either because test suites grow too large, because changes come in too fast, or because heavy or costly simulations are needed each time. Hence, regression testing is at times re-oriented towards selective re-testing for confidence on that newly added features or bug fixes do not have unwanted side-effects.

Specification-based testing Similar to model-based testing, specification-based testing aims to test the functionality of software according to its requirements (i.e., the model, instead of being specifically created for testing purposes, is the software specification). Product conformance with specification is checked by reviewing any reference documents (formal specifications, user manuals, etc.), stating the claims that are made about the system or application, testing them against the product, and reviewing the results for errors.

Hence, specification-based testing is dominated by the traceability of written specifications, and only the existence of a formal specification introduces the possibility of automating test generation.

State transition tables Also called *state transition testing*, state transition tables are one of many ways to specify state machines (just like state diagrams), which essentially consists in writing truth tables in which the inputs are the current software state, and the outputs include the next state, along with other outputs. The advantage of state transition tables is that they list all possible state-transition combinations, not just the valid ones, which may be needed in sensible, critical domains. Analysing all possible interactions allows to identify defects that may enable invalid paths from one state to another. Of course, this is only useful if the number of states and events is relatively small, since these tables become very large very quickly as their number increases.

Traceability matrix This method is used to validate the compliance of a product with customer requirements. The requirements are listed in a row of the matrix and the columns of the matrix are used to identify how and where each requirement has been addressed. The resulting matrix (also known as *requirements traceability matrix*) allows, then, to follow a top-level requirement into their implementation and test cases. Such traceability is a highly desirable property for any software system, and it is not only a requisite for verifying that requirements are fulfilled by the software, but also for identifying unaddressed or extra functionalities, and to assess how far a given test suite covers the requirements of the

functionalities to be tested. Correlation shown by the traceability matrix may be too loose (meaning there is no coupling between a requirements and a system module or test case) or too tight (reflecting a very complex relationship which should probably be simplified by splitting up either the requirement definition, the functionality implementation, or both).

Use case testing Use case testing approaches determine how to use employ cases to develop effective sets of test cases. Use cases are one of the most meaningful sources of information as far as user expectations about the software are concerned, so they represent not only the specification of a system, but also a practical way of testing its behaviour. Thus, this testing approach focuses the attention on the user rather than on the actions the application or system performs. This perspective can help identify test cases which other technique have more difficulties in seeing, but to be 'testable', use cases need to be stable, specific, complete, and correct. Analysing and identifying use cases paths, test cases can be derived to exercise those paths, after taking into account their equivalence, commonalities, and variance properties.

When we present the testing methodologies we propose later on in this chapter, we will be showing black box testing approaches in all cases.

6.1.3.3 Positive and negative testing

One last classification of testing approaches refers to them as **positive** and **negative**, with regard to the main strategy they adopt, either aimed to endorse that the software works as it is intended to, or else aimed to break the software showing proof that it does not work, breaks, or presents some unspecified behaviour.

Positive or *clean* testing is used mainly, if not exclusively, to check whether a given system, function or operation conforms to its initial design under regular conditions or ordinary interactions. On the other hand, the objective of negative or *dirty* testing is to purposely make the system, function, or operation fail, in order to check whether it responds soundly and is capable of handling anomalous situations appropriately. Simply put, the first philosophy checks for good behaviour in response to expected inputs, while the second checks for good behaviour even in presence of unexpected (wrong) inputs. It is clear that the ultimate goals of these two variants are completely different, but as different aims as they have, both approaches are complementary.

However, there is a school of thought in software testing that favours the idea that testing is not good enough unless it detects some defect. Admittedly, successful positive tests can never assure that a system or application performs correctly in every possible situation, they only grant that it does so in those situations represented by the tests. Still, verifying that something works as intended prior to submitting it to unusual conditions seems the most reasonable

approach. Positive testing answers the question of whether the product under test has the functionalities it was developed to fulfil. If the required properties and behaviours are not present in the software under normal circumstances, all other tests, especially negative tests, are irrelevant. Otherwise, it could be the ironic case in which a system is strong against malformed data, but it does not produce correct outputs for correct data. Indeed, negative testing is necessary in order to discover significant failures derived from the system response to external problems, software weaknesses, or malicious attacks, so both approaches need to be considered for genuinely effective testing.

In Section 6.3, we formulate different testing approaches which combine both positive and negative testing.

6.2 QuickCheck

It is generally accepted that software testing is as necessary as delicate a matter. In particular, designing good test sets is a non-trivial task, which can very easily be missed. Unconscious presumptions about the functionality of the element under test may leave important scenarios or possibilities out of the testing range/scope. That is why automatic test generation tools can be interesting and helpful products.

As a successful example, QuickCheck has proven itself as a useful testing assistant. Invented by Claessen and Hughes [122], it is an automatic tool for test-case generation and execution based on specifications. Working as a library, QuickCheck provides a simple domain-specific language (in a way, an extension of the same programing language it was developed in, Haskell) which allows the developer/tester to easily write down program specifications in the form of properties which should be satisfied. From those formal specifications, QuickCheck automatically generates, runs, and checks the results of a large number of random test-cases to see whether the properties hold.

While using software specifications for testing is not a novelty in itself, as we have seen in the previous section, directly deriving test cases from them is not a trivial task. However, functional programs are very well suited to automatic testing. Pure functions defined in functional languages are easier to test than those with side-effects from imperative paradigms, thanks to the absence of concern about state before and after execution. Besides, declarative source code has the power of the full language for expressing test cases due to properties such as pattern matching and symbolic interpretations. Compare, for instance, the following test case specifications in Java (using JUnit to manually test on a case-by-case basis) and Erlang (using QuickCheck to fully specify complete tests as properties):

```
// Imperative Java definition
public List listIntersection(List a, List b) {
  List intersection = new List();
  Iterator itB = b.iterator();
  while (itB.hasNext()) {
    Object element = itB.next();
    if (a.contains(element)) {
      intersection.add(element);
    }
  }
  return intersection;
}
// JUnit test
public void test() {
  List a = new List();
  List b = new List();
  List c = new List();
  a.add(1); a.add(2); a.add(3);
  b.add(3); b.add(4);
  c.add(3);
  assertEquals(c, listIntersection(a, b));
  ...
}


% Functional definition
list_intersection([], A) -> [];
list_intersection( A,[]) -> [];
list_intersection( A, A) ->  A;
list_intersection( A, B) -> [ X || X <- B, lists:member(X, A)].

% QuickCheck test
prop_intersection() ->
  ?FORALL({A, B}, {list(int()), list(int())},
    ?FORALL(E, lists:append(A, B),
       case lists:member(E, list_intersection(A, B)) of
          true  -> lists:member(E, A) andalso lists:member(E, B);
          false -> not lists:member(E, A) or not lists:member(E, B)
       end)).
```

In the first case, we contrast the output of the `listIntersection` function with its expected value, after a set of specific steps, and infer from the successful match the correctness of the function under test.

In the second one, however, we *declaratively* specify the behaviour that the `list_intersection` function is expected to fulfil at all times: whenever we take an element E from a list –which is the concatenation of two lists A and B to ensure we focus on relevant cases, if the element belongs to the intersection of A and B it means that it was already present in both A and B, not being present at least in one of them otherwise. This is precisely the formal definition of the intersection operation, which is tested with randomly generated values of lists of integers.

But, even if we perform imperative-like programming, the same testing philosophy can still be applied thanks to QuickCheck constructs based on algebraic laws, abstract models, and pre- and postconditions; the use of observational equivalence as result checking mechanism, and the explicit representation of state as data structures, when needed. Besides, with the QuickCheck specification language being embedded in the same programming context, the effort required to write properties is constrained to the specifications description, since there is no need to use any other descriptive language or formalism.

Undoubtedly, software specifications bring important benefits: they contribute to wider product understanding (no matter if they are written before the system implementation is done, as requirements formalisation, or afterwards, to serve as testing criteria) and better system documentation; being able to also use the same specifications for testing purposes only adds even more value to them. Of course, since source code and specifications are not derived from each other, rather they are independently written by developers or testers, there is no guarantee that the first is consistent with the latter. Testing one against the other with random inputs, however, is a way to empirically improve the confidence on that they actually are consistent. More systematic testing could seem more likely to be meaningful in this sense, but different studies show that, actually, this is not the case [123, 124].

Random testing does well in comparison with other black-box testing techniques, and even as well as they with only a small percentage of additional testing, with cost efficiency making up for it. However, improvement of random test cases effectiveness is possible if test data distribution can be influenced. To make that possible, QuickCheck also provides a test data generation language, which enables testers to control the distribution of test cases to conform to any desired and arbitrarily complex invariant.

There is a variety of QuickCheck implementations deriving from the original Haskell version, in different programming languages such us C++ [125], Java [126], or ML [127], among others. So far, there is only one commercially available version of QuickCheck, which is implemented in Erlang [128, 129] and has many features that the open source versions from the research community lack. This product is provided by Quviq and is referred to as Quviq QuickCheck. In the rest of the chapter we will show how we used Quviq Quickcheck as part of our strategy to improve the development life cycle of a software product, specifically for improving testing activities and results.

6.2.1 Property-based testing

As has been explained, Quviq QuickCheck is a powerful and versatile specification-based testing tool, aimed to test software against properties formulated in the programming language Erlang. Part of its strength comes from the capabilities it provides to easily write customised data generators and system

properties. In their simplest form, QuickCheck[1] properties are universally quantified formulae which read:

$$\forall \textit{ values in a certain set, a test depending on those values generates results meeting a specified condition} \quad (6.1)$$

[1] From now, we will refer to 'Quviq QuickCheck' just as 'QuickCheck'. Examples in this dissertation use version 1.13.

For example, the property that certain encode and decode functions on strings are each others inverse, can be expressed as

```
prop_encode_decode() ->
  ?FORALL(S, string(), decode(encode(S)) == S).
```

descriptively stating (thanks to the declarative syntax of a functional language like Erlang), that if we `encode` any `string S` and `decode` the result again, then we end up with the original string `S`. The terms `encode(S)` and `decode(S')` are invocations to the functions we want to test, and `string()` is a data generator which produces random lists of characters. QuickCheck provides a few basic data generators that can be used to write user-defined data generators:

```
alfanumeric_string() ->
  list(oneof([choose($0, $9), choose($A, $Z), choose($a, $z)])).

prop_encode_decode() ->
  ?FORALL(S, alfanumeric_string(), decode(encode(S)) == S).
```

Using these user defined generators, properties are tested automatically by QuickCheck against randomly generated test cases. On test success, the tool will just display an informative message:

```
> eqc:quickcheck(string_eqc:prop_encode_decode()).
..................................................................
..................................
OK, passed 100 tests
true
```

The number of test cases generated on each execution of QuickCheck is by default 100, but this can easily be adjusted to our needs.

```
> eqc:quickcheck(eqc:numtests(250,
                              string_eqc:prop_encode_decode())).
..................................................................
..................................................................
..................................................................
....................................................
OK, passed 250 tests
true
```

On test failure, the values of the `?FORALL`-bound variables are reported. For example, if we (wrongly) stated that encoding is the identity operation,

```
prop_encode() ->
  ?FORALL(S, string(), encode(S) == S).
```

and tested the property with QuickCheck, then we would see an output like:

```
> eqc:quickcheck(string_eqc:prop_encode()).
. Failed! After 1 test.
"n"
Shrinking... (1 times)
""
false
```

The testing fails, claiming that it is *false* that the specified property holds for any string input. Only one test was in this case enough to detect the failure, and such test input was the randomly generated string `"n"`, which turned out not to be its own encoding, hence representing a failing value of `S` or, in other words, a counterexample for the property we wanted to test.

Whenever a failing test case is found, QuickCheck *shrinks it* to a “minimal counterexample” to speed diagnosis [130]. In the previous example, the shortest character list which is not its own encoding is found to be the empty string (`""`), which is the minimal string we can build.

6.2.2 State machine testing

Of course, the greatest challenge to effectively use a tool like QuickCheck at full scale within a software development project, is expressing real specifications as testable properties. To make this process easier, QuickCheck provides high-level forms of specification on top of the universally-qualified properties seen in the previous section. These higher-level specifications are provided as domain-specific idioms embedded in the programming language, in other words, libraries whose API can be thought of as a new special-purpose language for expressing specifications of the chosen form.

One of these forms is state machines. Besides data generators and system properties, QuickCheck provides a mechanism to define a state machine that can be used to test state-full system behaviour in a very simple and structured manner. QuickCheck state-machine testing consists in specifying an initial state that can be modified as a result of state transitions. Transitions between states represent the operations (use cases) of the system to be tested. For each operation to be executable at a certain state, some conditions might have to be met, and some other would be true after the state transition is completed. Those conditions that need to be true before a certain operation

can be executed are called *preconditions*, and those conditions that should be fulfilled once the execution is finished are named *postconditions*. In summary, QuickCheck provides the means to define a state machine that will have a certain initial state, and a set of operations (transitions) with their corresponding pre- and postconditions, and how they affect the internal state.

Once the definitions of the state machine are written down, QuickCheck will automatically generate random sequences of state transitions from the initial state, each of them according to their preconditions (i.e., if a precondition is not true, that transition will not be eligible by QuickCheck to be the next step for the state machine in the sequence of randomly generated test cases), updating the internal state, and then checking their postconditions. To be able to decide whether the pre- and post-conditions are being fulfilled by a candidate transition for a test case, QuickCheck executes them *symbolically*.

Symbolic execution [131] is often used in frameworks for model-based testing as a means for searching for execution traces in an abstract model. Each execution path in the model (in the case of a state machine, each possible combination of sequential transitions from a given initial state) represents one possible program execution that can be used as test case. The simulated program execution normally uses symbols for variables rather than actual values, and those symbols are instantiated by assigning values to them when the test case is actually run.

Hence, after generating a test case using symbolic execution, QuickCheck proceeds with a real execution. Whenever a postcondition evaluates to false in the real exercise, the testing will stop and return an error. If QuickCheck runs all the test case transitions without running into any unexpected errors or conditions infringements, it will proceed to the next test case, and eventually exit informing of successful completion.

In short, the steps for using the QuickCheck state machine are these:

1. Define the state structure and the content for the initial state.
2. Specify the set of operations to test (and the generators for their input parameters).
3. For each operation, define its preconditions and postconditions, and how its execution modifies the content of the internal state.
4. Execute a set of automatic randomly-generated tests.

The UML activity diagram on Figure 6.1 explains in a graphical way how the QuickCheck state machine works. First of all, the structure of the state and its initial value needs to be defined. The internal state structure can be as simple or as complicated as needed. Depending on each case, we might need to store a list of identifiers, several sets of objects, a boolean flag, etc. Since QuickCheck puts no restrictions on the size or structure of the internal state,

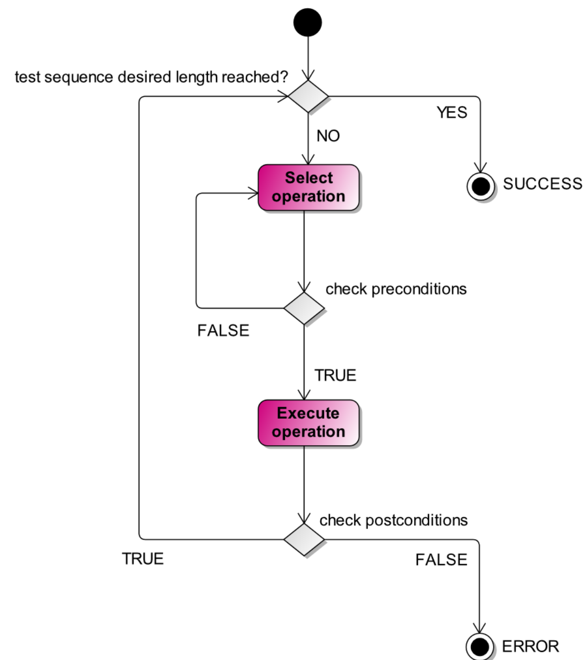


FIGURE 6.1. QuickCheck state machine

the test case developer has full freedom.

The internal state must hold the data we may need for the checks to be performed at the pre- or postcondition functions, and for the state change as well. According to this, a state definition will generally be a record with several fields:

```
-record(state, {fields_of_interest...}).
```

The initialisation of the state is done at the QuickCheck `initial_state/0` callback library function:

```
initial_state()->
  #state {fields_of_interest = initial_values(), ... }.
```

The state machine transitions (system, application, or component operations to be tested) are specified within the definition of the `command/1` callback function, which receives the internal state as argument:

```
command(S) ->
  oneof([{call, ?MODULE, operation_to_test1, [data_generator1()]},
         {call, ?MODULE, operation_to_test2, [data_generator2()]},
         {call, ?MODULE, operation_to_test3, [data_generator3()]},
          ...]).
```

This definition makes use of the QuickCheck generator `oneof/1`, which randomly selects an element from a list. Note that, as mentioned before, the previous code explicitly uses symbolic calls in order to ease the debugging when an error is actually found (as we will see in detail in Section 6.3.1),

```
{call, ?MODULE, function, Arguments}
```

instead of actual function calls like

```
?MODULE:function(Arguments)
```

Besides, QuickCheck allows to assign each transition with a probability of occurrence, as a positive integer:

```
command(S) ->
  frequency(
   [{10, {call, ?MODULE, operation_to_test1, [data_generator1()]}},
    {20, {call, ?MODULE, operation_to_test2, [data_generator2()]}},
    { 5, {call, ?MODULE, operation_to_test3, [data_generator3()]}},
    ...]).
```

meaning that the probability of occurrence of the corresponding operation in a generated test sequence is proportional to its associated weight (i.e., 10, 20, 5,...). This introduces very important advantages with respect to test case distribution management.

The best way to produce an exhaustive `command/1` function that includes all relevant operations is by adding them gradually, so we should start testing a small state machine with few transitions and check it is error-free before proceeding and considering more possibilities. It is very important to remember that each state machine operation should resemble an exact operation, use case, or functionality in our system, so that the state machine performs no extra work, or bears no extra responsibility apart from performing the actual transitions.

It might be the case that the actual operations in the system or module to be tested receive or return additional information that we are not interested in providing or that it is not needed at the postconditions. In these cases, to keep the state machine definition clean and simple, local wrapper functions for each operation to test can be defined and used as state transition operations. Wrapper functions can adapt both parameters and/or returned values for/from the original function, thus simplifying the interface between QuickCheck and the functions to test, and without modifying the code which is being tested.

For each operation/state transition, besides its preconditions and postconditions, a `next_state/3` callback function is necessary, which specifies how

the state transition (execution of the operation) affects the internal state. Of course, precondition, postcondition, and `next_state` functions must reflect the properties we want our system to have. Domain constraints should be identified and translated into these check points, always bearing in mind that actual fulfilment of those constraints must already be the responsibility of the code or component to be checked. That is, we need to confirm that those conditions are being assured, avoiding by all means restricting our test-cases more than the normal use of the application would.

```
precondition(S, {call,?MODULE,operation_to_test1,[parameters]}) ->
   (condition on [parameters] and S);
precondition(_State, _Call) ->
  true.
```

With regard to the effect of each function on the internal state, the `next_state` function must modify the internal state according to the executed transition:

```
next_state(S, R, {call,?MODULE,operation_to_test1,[parameters]})->
  NewState = #state {fields_of_interest = new_values,...};
next_state(State, _Result, _Call) ->
  State.
```

As for the postconditions, this is the place where important tests can be performed to check whether or not each operation result is what we expect. The `postcondition/3` callback function looks almost like the `precondition/2` callback function, with as many pattern matching cases as operations in the `command/1` definition:

```
postcondition(S,{call,?MODULE,operation_to_test1,[parameters]},R)->
   (condition on R and [parameters] and internal state S);
postcondition(_State, _Call, _Result) ->
  false.
```

Once we have built our QuickCheck state machine module, from the internal state definition to the postconditions implementation, it is time to let it run. When using QuickCheck state machine capabilities, this is done by invoking the testing of a generic property:

```
prop_state_machine() ->
  ?FORALL(Commands, commands(?MODULE),
    begin
      {_History, _State, Result} = run_commands(?MODULE,Commands),
      Result == ok
    end).
```

which calls QuickCheck function `run_commands/2` to execute automatically generated sequences of operations (`Commands`, produced by the generator `commands(?MODULE)`), and verifies an `ok` result.

Thus, QuickCheck provides easy-to-use mechanisms both to check software applications for properties that must hold and to check for systems or components behaviour adjusting to a certain set of state-transition rules, complying with preconditions and postconditions.

6.3 Testing ARMISTICE

In this last section of the present chapter, we will describe in detail the testing levels that were inspected in our case study, the testing techniques that were developed and put in practise, and the results and lessons that we learned from these experiences.

As mentioned, the ARMISTICE system has been in production for a few years now. Prior to that, different testing activities were conducted by both developers and regular users during several stages of the development. In the earliest development phases, verification was gradually performed by developers, in the form of unit testing of the functions and modules as they were individually being implemented. Studied in retrospective, these manual unit testing operations usually have revealed a natural tendency to fulfil boundary value analysis, as well as intuitive equivalence partitioning. Since a prototyping life cycle was being applied, no intensive integration testing tasks were required, because both client, server, and their communication protocol evolved in parallel, in an incremental and progressive fashion. Besides, some use-case testing was done by programmers themselves, specifically exploratory testing of application functionalities, as well as informal system and regression testing. These activities were complemented by the validation work carried out by final users from the moment the first working prototype was available, which was relatively soon. This early on-site deployment enabled the client to check the system in an exhaustive and ongoing acceptance testing process. We could say that the general philosophy behind the developer's testing efforts was aligned with the principles of negative testing while, on the other hand, users tended to be inclined towards positive testing, following what their daily routine activities would normally dictate.

Such a testing schema is not uncommon in software development cycles, but it is hardly ever complete and exhaustive. The fact that an application has been running daily without major problems is just weak empirical evidence of correctness. In order to provide a greater degree of confidence, and taking advantage of the application's core being implemented in Erlang, we decided to use QuickCheck to apply some model-based testing, in particular model-based automatically generated random tests involving different aspects of the system, from data types to business rules.

6.3.1 Data types verification

As a good starting point to enhance the testing processes that ARMISTICE had undergone, and consequently, the system reliance and faithfulness, we chose ARMISTICE data types. Data types are the smallest logic element that can be tested in most software applications; they are the components on which all other business objects are built upon.

When creating software, data types are the basic bricks. From application data types, business objects which implement business logic are built. Most of the time a programmer will use data types defined in library modules, therefore being tested by many users over many years. But sometimes, the appropriate data type is unavailable and has to be constructed from scratch. In this way, new basic bricks are created, and potentially used in many products in the future. Thus, data types are a key aspect, perfectly suitable to initially turn to when aiming to thoroughly test a software product, and the efforts devoted to test them definitely pay off.

As one can imagine when talking about such a complex software system as our case study RMIS, there are a number of data types implemented in ARMISTICE, and some of them (such as `logico` for booleans or `entero` for integers) are very similar to the basic data types found in Erlang, as in most commonly used programming languages. Upon these basic types, other data types are constructed, for example a data type `monetario` for representing amounts in different currencies.

One of the main reasons for the ARMISTICE development team to build anew not only complex, but customised basic data types, is to be able to have a uniform way of marshalling and unmarshalling values within the system. This is needed since, as we mentioned in Chapter 5, all communication between the ARMISTICE server and clients is performed via XML-RPC, an RPC protocol based on XML-formatted text messages. Marshalling and unmarshalling of Erlang terms into/from text strings occur any time the ARMISTICE server receives a request, and before any answer can be sent in response. For that reason, all data types have constructors to create a value from a string, and similarly, they all implement a function to convert a value to a string. Such operations are the basis of all communications with the client.

The method we present here is a structured methodology to follow when testing user-defined Erlang data types using QuickCheck. All ARMISTICE data types have been tested with this method.

6.3.1.1 Decimal data type

All ARMISTICE data types have the same structure: a value is represented by a record with the name of the type, and as many fields as required for the definition of the data type.

```
-record(logico, {value}).
-record(entero, {value}).
-record(monetario, {value, exchange, currency}).
```

Successful operations on a data type value return the new value wrapped in a tuple with `ok` as the first parameter:

```
> logico:new(true).
{ok, {logico, true}}
> entero:new(10).
{ok, {entero, 10}}
> monetario:new().
{ok, {monetario, {decimal, 10000000000000000},
                 {decimal, 10000000000000000},
                 {cadena,  "EURO"}}}
```

Or, similarly, using record notation:

```
> logico:new(true).
{ok, #logico{value = true}}
> entero:new(10).
{ok, #entero{value = 10}}
> monetario:new().
{ok, #monetario{value    = #decimal{value = 10000000000000000},
                exchange = #decimal{value = 10000000000000000},
                currency = #cadena{value = "EURO"}}
```

On the other hand, in case a data type operation results in an error, the value of the data type is represented by a tuple with first argument `error` and second argument an atom describing the cause of the error. Thus, a division by zero error with two `entero` values will not result in a crash:

```
> {ok, A} = entero:new(10).
{ok, {entero, 10}}
> {ok, B} = entero:new(0).
{ok, {entero, 0}}
> entero:divs(A, B).
{error, division_by_zero}
```

In this way, even failing operations on the server side are detectable at the client side.

As our leading example, and to illustrate the proposed method for testing data types with QuickCheck, we have selected the `decimal`, a data type for fixed point rational numbers, i.e., real numbers with a fixed number of decimal digits. This `decimal` data type is the basis for the `monetario` data type, which is used to represent sums of money, so it need not have the same range as floats.

The decimal data type is defined in a module called decimal.erl which exports a creator function named new in four flavours. As displayed in Fig. 6.2, the input to this constructor is either a single value or a two-element tuple (first component for the integer part, second the decimal part), with parameter values being either integer, float, or a string representation of one of the two. Besides, (only) when providing a single string value, it can contain commas as thousands separator and/or a single dot as decimal separator. The decimal separator cannot be used if the two-element tuple notation is used.

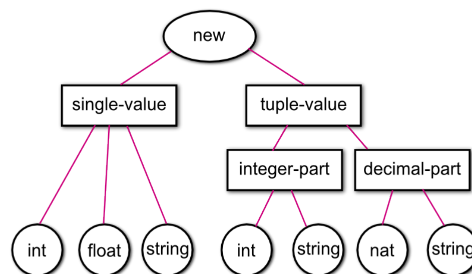


FIGURE 6.2. Decimal data type creation

Other decimal data type constructors provided include mathematical operators (sum, subtraction, product, division, negation, absolute value, maximum, and minimum) and relational operators (such as 'greater than', and 'less than or equal'). As mentioned, unlike the Erlang floats, this data type stores just a fixed number of decimal digits, rounding values when necessary.

Now, when first using QuickCheck, the provided documentation includes all the information about the libraries functionalities but, understandably, no advice is given about the right methodology to be applied. Thus, when an initial attempt is performed, it is very likely that too naïve an approach is followed. To begin with, it is necessary to define a data generator (in this case, for the decimal data type), to enable QuickCheck to create random instances for test case input, and at least one property that represents the expected characteristics and expresses the desired features (again, of the data type). A simple approach to generate decimals would be to define the following QuickCheck generator in which the function new[2] is applied to an arbitrary integer and an arbitrary positive integer as, respectively, the integer and decimal parts of the intended decimal:

[2] To enhance reading, operations have been simplified to ignore the previously mentioned ok tag from their results

```
decimal() ->
  ?LET(Tuple, {int(), nat()}, decimal:new(Tuple)).
```

where int() and nat() are data generators for integer and natural numbers provided by the QuickCheck libraries, and ?LET is a primitive that binds a variable (Tuple) to a value obtained from a generator (or combination of gen-

erators) before using it on the `?LET` body expression (`decimal:new(Tuple)`). This generator ignores three of the four ways to construct a `decimal` but, since it seems to be able to produce all possible `decimals`, it might look perfectly suitable to the non-expert eye.

In addition, the kind of properties that can easily come to mind in a context like this are, for example, a check that the sum operator is actually commutative:

```
prop_sum_comm() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
          decimal:sum(D1, D2) == decimal:sum(D2, D1)).
```

If we use QuickCheck to check such a property, with successful results, it would mean that the specified condition holds for thousands of randomly generated test cases or, in other words, that among thousands of randomly generated test cases, QuickCheck has been unable to find one that did not fulfil the condition and would serve as counterexample:

```
> eqc:quickcheck(eqc:numtests(10000,
                              decimal_eqc:prop_sum_comm())).
(...)
OK, passed 10000 tests
true
```

To reassure the confidence in QuickCheck generating evenly assorted tests when executing the data generators, there are a few library utilities to inspect the distribution of the generated input data, such as `collect/2` or `measure/3`:

```
prop_sum_comm() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
          collect({decimal:get_value(D1), decimal:get_value(D2)},
                  decimal:sum(D1, D2) == decimal:sum(D2, D1))).

> eqc:quickcheck(decimal_eqc:prop_sum_comm()).
(...)
OK, passed 100 tests
3% {0.0,0.0}
1% {20.29,-17.15}
1% {20.15,-25.15}
1% {20.0,14.13}
1% {19.15,20.0}
1% {15.1,5.18}
1% {11.26,-30.25}
...
true
```

```
prop_sum_comm() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
    measure("Inputs", [decimal:get_value(D1),decimal:get_value(D2)],
                        decimal:sum(D1, D2) == decimal:sum(D2, D1))).

> eqc:quickcheck(decimal_eqc:prop_sum_comm()).
(...)
OK, passed 100 tests
Inputs:     Count: 200    Min: -30.2     Max: 29.21
            Avg: -1.1482000000000006    Total: -229.6400000000001
true
```

As seen above, after a successful execution, `collect/2` classifies and displays the distribution of input data that has been used, providing interesting feedback in order to check whether the space of possible test cases is being evenly covered. Respectively, `measure/3` also collects some information about the specified values (in the previous case the operation arguments), and shows statistics such as the minimum, average, and maximum values.

In this example, commutativity of `sum` seems to be fairly tested and to be indeed commutative, but this kind of 'innocent testing' leaves us facing questions such as: which other properties need to be checked?, or when have we added sufficiently many properties, so that we can be satisfied with testing coverage? It certainly does not seem enough to test just one property for each operator, but exhaustively testing all and every property we may think of for each one of them is clearly impractical. These test completeness-related issues actually represent the arguments why this approach can be qualified as naïve, and sustain the claim for a more structured and defined process.

6.3.1.2 Model for decimal data type

Using knowledge from the field of mathematics and formal methods [132, 133], we can state that creating a model of the testing subject (i.e., the data type) could help in deciding whether we have created enough properties for it or not. Once a model is established, it is necessary to show that each operation on `decimals` can be emulated in the model, by model operations.

Thus, we formulate an injection $[\![\circ]\!]$ from the `decimal` data type into our model, such that $\forall d_i, d_j \in$ `decimal`,

$$
\begin{aligned}
[\![sum(d_i, d_j)]\!] &\equiv [\![d_i]\!] + [\![d_j]\!] \\
[\![subs(d_i, d_j)]\!] &\equiv [\![d_i]\!] - [\![d_j]\!] \\
[\![mult(d_i, d_j)]\!] &\equiv [\![d_i]\!] * [\![d_j]\!] \\
[\![divs(d_i, d_j)]\!] &\equiv [\![d_i]\!] \,/\, [\![d_j]\!] \\
[\![lt(d_i, d_j)]\!]_l &\equiv [\![d_i]\!] < [\![d_j]\!] \\
&\ldots
\end{aligned}
\tag{6.2}
$$

In general, we may need to implement a model with all these operations. In this case, though, as our model we can use the standard Erlang implementation of floating point numbers (in itself built upon a C definition that implements the IEEE 754-1985 standard [134]). In other cases, though, a simpler model than the data type itself can be relatively easy to implement, for instance not caring about efficiency and leaving out optimisations.

A simple injection is hence chosen for this particular example, namely mapping ARMISTICE's `decimals` to Erlang floating point numbers. In fact, this injection function was already present in the `decimal` module source code under test, in the formerly seen `get_value/1` function:

```
decimal_model(Decimal) ->
  decimal:get_value(Decimal).
```

which makes use of built-in functions in the language [135] to translate from the internal float representation that the `decimal` module uses (strings) to floating point numbers.

Note that the last equation shown in expression 6.2 includes a different model ($[\![\circ]\!]_l$), used for interpreting the result of the function `lt(D1,D2)`, since that result is not a `decimal` but a boolean value (i.e., `logico`). The model for the `logico` data type simply maps values to Erlang booleans and the injection is also already present in the corresponding module `logico.erl`,

```
logico_model(Logico) ->
  logico:get_value(Logico).
```

Now the QuickCheck properties to check whether ARMISTICE's `decimal` implementation is equivalent to the Erlang floating point implementation look like:

```
prop_sum() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
          decimal_model(decimal:sum(D1, D2)) ==
                           decimal_model(D1) + decimal_model(D2)).

prop_lt() ->
  ?FORALL({D1, D2}, {decimal(), decimal()},
          logico_model(decimal:lt(D1, D2)) ==
                         (decimal_model(D1) < decimal_model(D2))).
```

If one such property is created for each operation defined in the data type, then by checking each of them for a large number of random inputs, we would gain confidence that the data type operations have been sufficiently tested.

When the previous `prop_sum/0` property was first tested for ARMISTICE's `decimals`, it immediately resulted in a failure:

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
....Failed! After 5 tests.
{{decimal,1000000000000000}, {decimal,11000000000000000}}
false
```

After only five tests, QuickCheck found a counterexample against the equivalence between adding two `decimal` values and adding the corresponding two floats. It is worth remembering again that these testing activities were performed on the system when it had already been in production for a few years, hence the bugs that were found using them had not been detected or diagnosed by other means.

The counterexample values reported back by QuickCheck (accessible again if needed via the function `counterexample/0`), which are aimed to be used as input to reproduce the failure, and thus help locate the problem, are shown in their internal representation. This representation can be hard to understand by someone else who is not familiar with ARMISTICE's `decimal` data type implementation, so it is not desirable to deal with when performing testing activities. The situation would be even worse for more complex data types, and for a trained QuickCheck user, the values would be even surprising, since rather small integer values are expected as first test case scenarios, rather than values with 15 or more zeroes. The fact that the test actually failed with 1 and 1.1 is only directly obvious to the developer of the `decimal` data type.

Additional reasons further discourage relying on the internal representation of the data type. Apart from being hard for others than the developer of the module to understand, the implementation may change due to refactoring or optimisation, so we want tests to depend on source code details as little as possible, same as we want to avoid implementations depending on the kind of tests that are meant to be performed afterwards. Moreover, the internal representation is only the final result of a computation constructing the data structure, which may even vary for the same value depending on the way it is indeed constructed. Hence, we would rather stick to a black box testing approach and just know which steps were followed for the construction of the value (instead of its 'physical' structure), since that is the most revealing information concerning an observed failure.

Therefore, previous QuickCheck generators are modified to work with **symbolic** values rather than **real** values. This means that the generator functions will be set to produce symbolic representations of objects creation (in this case, `decimal`), which will be evaluated when needed, in place of actual values. So, the `decimal` generator for testing is rewritten to:

```
decimal() ->
  ?LET(Tuple, {int(), nat()}, {call, decimal, new, [Tuple]}).
```

thus generating a symbolic call to `decimal:new(Tuple)` in the form of a tuple `{call, ?MODULE, ?FUNCTION, ?ARGUMENTS}` with tag `call`, Erlang source code module `?MODULE` (i.e., `decimal`), function name `?FUNCTION` (i.e., `new`), and list of arguments `?ARGUMENTS` (i.e., `[Tuple]`), where `Tuple` is itself a tuple of generators `{int(), nat()}`, which will be replaced by the corresponding automatically generated values at execution time) instead of actually performing the call.

Of course, QuickCheck testing properties need to be changed accordingly and introduce the evaluation of symbolic values using `eval/1`, a standard QuickCheck function:

```
prop_sum() ->
  ?FORALL({SD1, SD2}, {decimal(), decimal()},
          begin
            D1 = eval(SD1),
            D2 = eval(SD2),
            decimal_model(decimal:sum(D1, D2)) ==
                             decimal_model(D1) + decimal_model(D2)
          end).
```

With these modifications, the failure is now reported back by QuickCheck as:

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
........Failed! After 9 tests.
{{call,decimal,new,[2,1]}, {call,decimal,new,[2,2]}}
Shrinking..(2 times)
{{call,decimal,new,[0,1]}, {call,decimal,new,[0,2]}}
false
```

Now it is much easier to see that the problem is detected by using values 2.1 and 2.2. Besides, the same error can be reproduced using values 0.1 and 0.2, as pointed out by the result of the automatic shrinking process performed by QuickCheck, which on error detection looks for the smallest equivalent failing case. The automatic shrinking affects the values in the symbolic calls and tries to narrow down the bug search by determining simpler input which will still crash the property in the same way.

Back to the ARMISTICE `decimal` data type testing, the detected problem is explainable according to an overlooked difference between the `decimal` definition and the model used for testing, rather than a bug in the actual implementation. Conforming to the IEEE 754-1985 standard [134], Erlang float values present an unavoidable rounding error, empirically demonstrated just by performing simple calculations:

```
> (0.1+0.2) == 0.3.
false
```

```
> (0.1+0.2) - 0.3.
5.55112e-17
```

On the other hand, ARMISTICE's `decimals` implementation use and store a maximum of 16 decimal digits, which immediate consequence is that there is not always an exact bijection between a `decimal` and Erlang floats. In other words, there is not always an exact representation of each `decimal`. However, ARMISTICE computations with `decimals` are performed on the `decimals`, and conversions are only required for the testing purposes we have seen. Hence, we are more than satisfied with an approximate equality:

$$a \approx b \quad \Leftrightarrow \quad |a| - |b| < \epsilon_{abs} \ \wedge \begin{cases} \dfrac{|a| - |b|}{|a|} < \epsilon_{rel} & \text{if } |a| > |b| \\[2ex] \dfrac{|a| - |b|}{|b|} < \epsilon_{rel} & \text{if } |a| < |b| \end{cases} \tag{6.3}$$

The previous equivalence relation is defined with respect to two maximum tolerance levels: an absolute error value (ABS_ERROR, ϵ_{abs}), which measures how different two floats are; and a relative error value (REL_ERROR, ϵ_{rel}), which takes into account not only the values themselves, but also their magnitudes [136]. Note that the implemented equivalence function `equivalent/2` divides the difference by the maximum of the absolute values of the two floats, ensuring that the maximum is never zero (unless they both are zero, in which case the absolute error value is used).

```
-define(ABS_ERROR, 1.0e-16).
-define(REL_ERROR, 1.0e-10).

equivalent(F, F)  ->
  true;
equivalent(F1,F2) ->
  if (abs(F1-F2) < ?ABS_ERROR) -> true;
     (abs(F1) > abs(F2))       -> abs( (F1-F2)/F1 ) < ?REL_ERROR;
     (abs(F1) < abs(F2))       -> abs( (F1-F2)/F2 ) < ?REL_ERROR
  end.
```

To set the reference error values, we use the knowledge that the `decimal` data type in ARMISTICE has, as we have already mentioned, 16 digits precision (hence, the value of ABS_ERROR, $\epsilon_{abs} = 10^{-16}$) and agree to a 99.9999999999% accuracy (hence, the value of REL_ERROR, $\epsilon_{rel} = 10^{-10}$). Just a minor change in the QuickCheck specification is necessary to introduce the new equivalence function:

```
prop_sum() ->
  ?FORALL({SD1, SD2}, {decimal(), decimal()},
          begin
            D1 = eval(SD1),
            D2 = eval(SD2),
            equivalent(decimal_model(decimal:sum(D1, D2)),
                       decimal_model(D1) + decimal_model(D2))
          end).

prop_lt() ->
  ?FORALL({SD1, SD2}, {decimal(), decimal()},
          begin
            D1 = eval(SD1),
            D2 = eval(SD2),
            logico_model(decimal:lt(D1, D2)) ==
                        (decimal_model(D1) < decimal_model(D2)))
          end).
```

Finally, these properties pass thousands of randomly generated test cases.

6.3.1.3 Generators to cover data structure

The natural progression of the testing activity would proceed by defining similar properties, one for each operation on the data type. But, even though it might seem so, this will not mean that the `decimal` data type has been tested completely.

As explained earlier, ARMISTICE's decimals can be constructed in a number of ways, reflected in the different inputs the `new/1` constructor accepts (cf. Fig. 6.2). However, the approach followed so far ignores the distinct building flavours in the belief that any decimal can be produced with the variant chosen by the `decimal/0` generator, whose code is kept small and simple. Common sense dictates, though, that testing is not only about producing possible output values, but exercising possible input value combinations. Doing otherwise may result in missed opportunities to detect errors, as we will see right away.

In addition, it should also be considered that operations that modify the data structure may actually violate invariants or invalidate properties that were true right before applying them. For instance, imagine a data type `ordered_set` in which elements are stored in a sorted list; a set obtained from the union of two sets may invalidate that invariant if the union operation is faulty. As a consequence, deletion on such a union can fail even if set element removal is working perfectly under normal conditions. Besides, if a property involving computing set union has already been defined, even code coverage techniques will most likely *not* reveal such a testing deficiency: testing might involve all source code belonging to different operations, but coverage analysis does not usually reveal that testing is not enforcing execution of *combinations* of operations,

meaning that a 100% line coverage can hardly be assimilated to testing of all relevant cases.

Similarly, in our case study, there is part of the data structure that has not been actually tested. We want to test things such as, for example, a multiplication of two `decimals` where operands need not have been obtained via the `new/1` creator:

```
decimal:mult(decimal:new("12,837.12"),
             decimal:sum(decimal:new(12), decimal:new({13,4}))).
```

To do so, a **recursive generator** is proposed, to produce arbitrary nesting of `decimals` as arguments of the QuickCheck testing constructor. The depth of the recursion is determined by QuickCheck such that small values are tried first, slowly growing as long as no errors are being detected. Access to the parameter that controls recursion depth is granted via the macro `?SIZED`:

```
decimal() ->
  ?SIZED(Size, decimal(Size)).

decimal(0) ->
  {call, decimal, new, [oneof([int(),
                               real(),
                               decimal_string(),
                              {oneof([int(), list(digit())]),
                               oneof([nat(), list(digit())])}])]};
decimal(Size) ->
  Smaller = decimal(Size div 2),
  oneof([decimal(0),
         {call, decimal, sum, [Smaller,Smaller]},
         {call, decimal, mult,[Smaller,Smaller]}]).
```

Not only have all the `new/1` function variants been included in this improved generator as ways of creating `decimal` data structures, but also some operators such as `sum` and `mult`. Recursion termination is granted by the reduction of the `Size` parameter at each step of the process[3].

[3] Additional code of `decimal` auxiliary generators can be seen in [137].

Recursive generators like this can generate symbolic calls that cover the whole data structure, as we can check using the QuickCheck `sample/1` utility:

```
> eqc_gen:sample(decimal_eqc:decimal()).
{call,decimal,sum,
   [{call,decimal,sum,
       [{call,decimal,mult,
          [{call,decimal,new,[{11,"4003351"}]},
           {call,decimal,new,["-930764"]}]},
        {call,decimal,new,[-2.35986]}]},
    {call,decimal,new,[1.64783]}]}
```

Re-running the previously defined property for the sum operator, thousands of tests are successfully performed in a few seconds, increasing our confidence in the implementation. But the testing activity will not be complete until properties for all operators have been defined, and all operations which produce new decimal data type structures as a result are included as part of the recursive generator. However, doing so with the multiplication resulted in a new failure:

```
prop_mult() ->
  ?FORALL({SD1, SD2}, {decimal(), decimal()},
          begin
            D1 = eval(SD1),
            D2 = eval(SD2),
            equivalent(decimal_model(decimal:mult(D1, D2)),
                       decimal_model(D1) * decimal_model(D2))
          end).

> eqc:quickcheck(decimal_eqc:prop_mult()).
...............Failed! After 16 tests.
{{call,decimal_eqc,sum,
   [{call,decimal_eqc,sum,
      [{call,decimal_eqc,new,["+1"]},
       {call,decimal_eqc,new,[2.36314e+4]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,[-5]},
       {call,decimal_eqc,new,[-9.61993e+5]}]}]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["74.4"]},
       {call,decimal_eqc,new,[{"-6,179","40"}]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["47"]},
       {call,decimal_eqc,new,["-467,725.079"]}]}]}}
Shrinking..............................(31 times)
{{call,decimal_eqc,sum,
   [{call,decimal_eqc,sum,
      [{call,decimal_eqc,new,["+0"]},
       {call,decimal_eqc,new,[0.00000e+0]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,[1]},
       {call,decimal_eqc,new,[10.1400]}]}]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["00.4"]},
       {call,decimal_eqc,new,[{"-0,000","40"}]}]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["40"]},
       {call,decimal_eqc,new,["-000,000.078"]}]}]}}
false
```

As we can see from this example, the failing test case contains a fairly large expression. The shrinking procedure reduces the test case significantly, but there are still a number of terms that a human tester would reduce further. For

example the sign could be removed from `{call,decimal,new,["+0"]}`, but even better, the whole term could be removed. Another simplification could be used for the string `"-000,000.078"`, where six zeroes can be reduced to one, at least if the value is important and not the actual structure of the string. The reason why these terms are not shrunk any further lays in the definition of our generators, and in the fact that QuickCheck automatic shrinking is focused on value complexity rather that on value structure, which is user-defined. We will first try to improve this shrinking behaviour, before diagnosing the problem.

6.3.1.4 Improving shrinking

To allow testers to improve situations in which automatic shrinking is not good enough, QuickCheck offers a couple of macros `?SHRINK` and `?LETSHRINK` which can be used to define customised shrinking rules, manually providing shrinking alternatives which are applied before the built-in ones.

[4] For details on the shrinking improving process, again refer to [137].

With these rules for shrinking added to the recursive generators[4] we increase the simplicity of the returned failing test case without losing accuracy.

In addition to simplifying the terms, being able to see the difference between the value returned by the implementation under test and the value computed in the corresponding model is also very helpful when diagnosing a failure. The QuickCheck `?WHENFAIL` macro allows to process additional information (first argument) whenever the second argument (generally, the property) evaluates to false:

```
(...)
decimal(Size) ->
  Smaller = decimal(Size div 2),
  oneof([decimal(0),
         ?LETSHRINK([D1, D2], [Smaller, Smaller],
                    {call, decimal, sum, [D1, D2]}),
         ?LETSHRINK([D1, D2], [Smaller, Smaller],
                    {call, decimal, mult, [D1, D2]})]).

prop_mult() ->
   ?FORALL({SD1, SD2}, {decimal(), decimal()},
           begin
             D1 = eval(SD1),
             D2 = eval(SD2),
             Real   = decimal_model(decimal:mult(D1, D2)),
             Model = decimal_model(D1) * decimal_model(D2),
             ?WHENFAIL(io:format("Real ~p  nModel ~p  n", [Real,
                                                          Model]),
                       equivalent(Real, Model))
           end).
```

Checking this re-defined property against the same example (which can be done thanks to the QuickCheck `recheck/0` function) showed a difference in

real vs. model outcome by a factor 10. Running QuickCheck a few additional times always returned the same factor 10 difference.

```
Shrinking.....................(51 times)
{{call,decimal_eqc,new,[10.1400]},
 {call,decimal_eqc,sum,
   [{call,decimal_eqc,new,["0.4"]},
    {call,decimal_eqc,mult,
      [{call,decimal_eqc,new,["47"]},
       {call,decimal_eqc,new,["-0.078"]}]}]}}
Real -331.172
Model -33.1172
false
```

Thanks to this information, the difference was rather quickly identified as an error in the `decimal` module implementation: the carrier was incorrectly propagated. The problem arose when values were rounded to ignore the least significant digits, which are not to be stored in ARMISTICE `decimals` implementation, as previously said. In such cases, a rounding operation was considered for the last decimal digit to be stored, but no carrier was being taken into account to be propagated to the left. Instead, if the last decimal was to be modified (rounded) and this digit turned out to be a 9, then the 9 was erroneously replaced by a 10. For instance, when rounding a large number like

$$481.51623424815162\mathbf{39}|942$$

to sixteen significant digits (same as the real implementation), we should obtain

$$481.51623424815162\mathbf{40}|000$$

But, instead, the erroneous code was replacing it by

$$481.51623424815162\mathbf{31}|000$$

Since the internal representation of `decimals` is a sequence of digits with a fixed number (16) of decimals, this longer sequence was then interpreted as:

$$4815.1623424815162310$$

Strangely enough, this rounding error had been in the code for several years without being found a problem. However, after diagnosing it, it was actually

found related to some obscure error reports from the ARMISTICE's users, to that date unsolved.

6.3.1.5 Well defined generators

After correcting the code, properties for addition and multiplication operators passed thousands of generated test cases. Creating, as remarked, additional properties for the yet untested operations (subtraction, division,...) takes hardly any effort having the previous testing structure in place and following the same steps.

However, it is highly convenient to re-check once more all defined properties after each incorporation, since new errors can be revealed by the new combinations. In this case, it is the property for addition which fails again, crashing in the evaluation of a generated value:

```
> eqc:quickcheck(decimal_eqc:prop_sum()).
............Failed!
After 13 tests.
Shrinking....(4 times)
Reason:
{'EXIT',{{error,decimal_error},
          [{decimal_eqc,'-prop_subs/0-fun-0-',1},
           {eqc,'-forall/2-fun-4-',2},
           ...]}}
{{call,decimal,new,[0]},
 {call,decimal,divs,
   [{call,decimal,new,[{0,[]}]},
    {call,decimal,new,["0"]}]}}
false
```

Specifically, the failure case reveals that when adding two values, a simple one (i.e., `{call,decimal,new,[0]}`) and a complex one obtained as a result of dividing two simple values (i.e., `{call,decimal,new,[{0,[]}]}` and `{call,decimal,new,["0"]}`), the process fails if the latter is a division involving a zero denominator. Indeed, as pointed out by the shrinking process counterexample, the second operand is the result of a division by zero. In other words, a symbolic value that does not correspond to a real value has been generated as input for the property.

While we absolutely want to test that the division operation implementation detects and handles expected error situations such as division by zero nicely, this is done in the property for division. There, we test that a division in the model results in the same value as a division of `decimal` values, including behaviour on anomalous situations. Hence, division by zero in the model should generate an exception similar to to the one raised by the implementation:

```
prop_divs() ->
  ?FORALL({SD1, SD2}, {decimal(), decimal()},
          begin
            D1 = eval(SD1),
            D2 = eval(SD2),
            case equivalent(decimal_model(D2), 0.0) of
              true ->
                {'EXIT', _} = (catch decimal_model(D1)/
                                     decimal_model(D2)),
                {error, _}  = decimal:divs(D1, D2);
              false ->
                equivalent(decimal_model(decimal:divs(D1, D2)),
                           decimal_model(D1) / decimal_model(D2))
            end
          end).
```

With such a property, we already check that division by zero is an exception case. Now what we want is to avoid generating these exception cases as part of the QuickCheck generators activity. Instead, we only want to generate *well defined* symbolic values, meaning that such symbolic values do not raise an exception when evaluated.

To do so, a simple, generally applicable concept is used. An auxiliary test function `defined/1` is written, which evaluates an automatically generated symbolic value and catches potential exceptions; the symbolic value is said to be *defined* only if no exception occurs. We rely on the fact that the majority of the symbolic values will not raise an exception when evaluated, and we introduce a `well_defined/1` complementary generator to keep generating values until we find a defined one:

```
defined(E) ->
  case catch eval(E) of
    {'EXIT', _} -> false
    _Value      -> true;
  end.

well_defined(G) ->
   ?SUCHTHAT(E, G, defined(E)).
```

These generators are now used in our QuickCheck specification for ARMISTICE `decimals`:

```
decimal() ->
  ?SIZED(Size, well_defined(decimal(Size))).
```

This way the generation process is filtered and never produces faulty values. Admittedly, this might seem biased at first sight, but it is actually not, as long as

we ensure that we check each operation in a specific property that deals with the acknowledged faulty situations that are expected around such operation. The only case missing is the generation itself, where we need to check whether specific inputs crash the `new/1` operation. For that purpose, an additional property is added to test that generating base values always succeeds (as expected). In addition, we also check that translating the created `decimal` into the model always produces a valid value:

```
prop_new() ->
  ?FORALL(SD, decimal(0), is_float(decimal_model(eval(SD)))).
```

At this point, only one last property remained untested, which was part of the motivation for introducing the customised ARMISTICE data types. Since client-server communication is performed using XML-RPC, marshalling to and from strings was a requirement that we now must verify: for each and every `decimal` data structure that is produced in any of the different possible ways, converting it to a string and performing the reverse operation needs to be idempotent:

```
prop_decimal_string() ->
  ?FORALL(SD, decimal(),
          begin
            D = eval(SD),
            decimal:new(decimal:to_string(D)) == D
          end).
```

With this last definition, and finally succeeding in passing hundred thousands of tests for each property in the `decimal` implementation, we have thoroughly tested the data type, concluding its verification process.

6.3.2 Integration testing

The kind of testing described in the previous section can be classified as black-box unit testing in which specific pieces of software (i.e., Erlang modules) are tested on their own to see if they behave as expected. A large number of automatically generated test cases are presented as input, and outputs are inspected on the basis of descriptive properties.

On the next testing level, we use QuickCheck in a different way. In ARMISTICE, as likely in any client-server application of a certain magnitude, different teams (or at least different people) develop in parallel the user client and the business logic functionalities. In this traditional architecture, according to the Model-View-Controller pattern, the user interface and the back-end server are both connected thanks to a third component, usually referred to as *controller* (cf. Fig. 6.3).

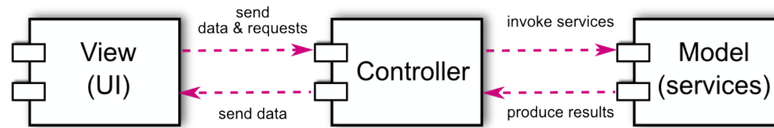


FIGURE 6.3. Model-View-Controller architectural pattern.

The controller receives clients petitions and dispatches them, invoking the corresponding model services. Upon server reply, the controller forwards the responses to the requester. Ideally, controllers should embody no real knowledge about server structure or distribution, and should only perform invocation format conversions, if needed.

On the other hand, clients, controllers, and most of all, servers, are conveniently structured if additional patterns such as Layers (cf. Fig. 6.4) are used. The Layers pattern allows to present a unique (or selected) access point(s) to third parties, while enabling software hierarchisation, isolation of future implementation changes, ease of management duties, as well as refactorisation and optimisation activities. In such fractionated systems, integration of the different components, at all levels, is a fundamental aspect to test. Once all components have been tested for integrity, they need to be tested for integration. Seamless interaction between all the pieces that conform an application is crucial for the good operation of the entire software, since its global functionality depends on such intercommunication to be virtually error-free (assuming each part has been already subject of individual testing procedures).

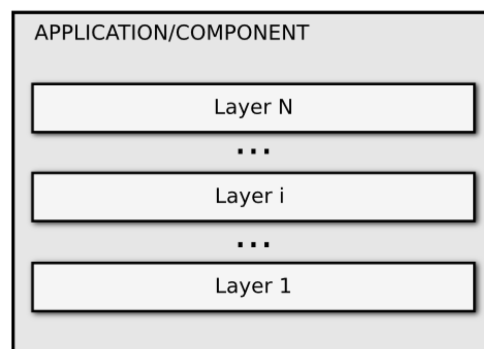


FIGURE 6.4. Layers architectural pattern.

This scenario applies of course not exclusively to client-server layered architectures or MVC-structured systems. Modern systems are more and more frequently designed and implemented as a set of separate components that

work together to offer some service or provide some functionality. In all those situations, systems, and applications, integration is an essential aspect to test.

In the following pages, we will explain in depth a method to perform integration testing that has been exercised on ARMISTICE using QuickCheck, the same automatic testing tool we have used to perform unit testing.

6.3.2.1 State machine based methodology

Bearing in mind that we want to test integration between ARMISTICE's clients and server, the list of relevant test cases is precisely the list of requestable services, this is, the set of exported functionalities that form the public interface of the ARMISTICE server. In a good software design, an application provides one or more facades [74] with details for invoking the different interface functions. The testable set would then generally be the list of procedures exported by each single system facade. Thus, integration testing will consider different sets of related use cases at a time, based on the presence of access facades, or else functionality, as classification or partition criterion.

To illustrate the use of QuickCheck for integration testing purposes, we have chosen the set of ARMISTICE use cases in Figure 6.5, belonging to the risk group and risk object management facades. As one can imagine, even the execution of these management operations on these fundamental business objects involves the invocation of different functions in several internal modules in the different internal layers of the application server. The facades play an intermediate role, offering a unique interface function to the clients, and translating each request into the necessary sequence of function invocations to provide the corresponding functionality. Return values from intermediate calls will be used as input data for others, until a final result is obtained, and sent as the service response. Since any operation in ARMISTICE needs to be performed by a specific user (that must be logged in), we have added to the set of use cases the user log in and log out functionalities.

To use the QuickCheck state machine, we begin by defining the structure and contents of its internal state, as well as the initial value for such state. The data stored as internal state of the testing state machine can be as simple or as complicated as needed, and it will be useful to favour the generation of transitions (command invocations) on related input values, which translates into more realistic, and thus interesting, test sequences. Depending on the set of functionalities to test (i.e., the input parameters of the interface functions to be tested) we might need to store a list of object identifiers, several sets of objects, a boolean flag, etc. Stored values can be later used on checks performed at the `precondition/2`, the `postcondition/3`, and the `next_state/3` functions. In integration testing, as we will see, the most important checks will take place in the postconditions.

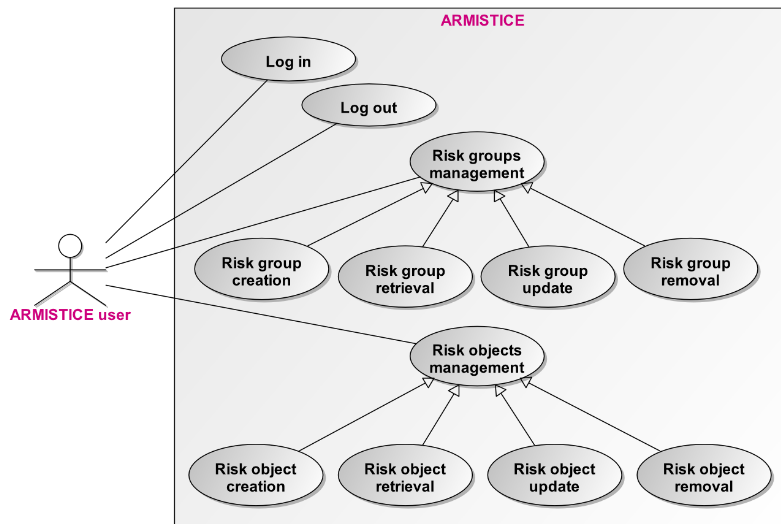


FIGURE 6.5. Risk groups and risk objects management use cases.

In our ARMISTICE case study, the state will be composed of:

- user sessions, data about users that have successfully logged in,
- groups, data about the risk groups that have been created, and
- objects, data about the risk objects that have been created.

According to this, the definition of the state will be:

```
-record(state, {user_sessions, groups, objects}).
```

where each element in the `user_sessions` list field is a unique `SessionID` identifying the user, each element in the `groups` list field is a unique `GroupID` identifying a risk group, and finally each element in the `objects` list field is an `ObjectID` identifying a unique risk situation.

The initialisation of the state is done at the `initial_state/0` QuickCheck callback function:

```
initial_state()->
  #state{user_sessions = [],
         groups = [],
         objects = []}.
```

The `user_sessions` represent the valid session ID of users that have logged in but have not logged out, meaning that they can invoke services (regardless of them being allowed to actually perform the operations or not, based on their user profile). The `groups` and `objects` lists start empty even though

they could have been initialised using already stored data from ARMISTICE's persistent storage.

Next, the state machine transitions are specified within the definition of the library callback function `command/1`, which will be the operations or exported functions to be tested:

```
command(S) ->
  frequency([{ 1, {call, ?MODULE, login, [user()]}},
             {10, {call, ?MODULE, new_risk_group,
                                  [oneof(S#state.user_sessions),
                                   group_name()]}},
             {50, {call, ?MODULE, find_risk_group,
                                  [oneof(S#state.user_sessions),
                                   oneof(S#state.groups)]}},
             { 5, {call, ?MODULE, update_risk_group,
                                  [oneof(S#state.user_sessions),
                                   oneof(S#state.groups)]}},
             { 2, {call, ?MODULE, delete_risk_group,
                                  [oneof(S#state.user_sessions),
                                   oneof(S#state.groups)]}},
             {50, {call, ?MODULE, new_risk_object,
                                  [oneof(S#state.user_sessions),
                                   oneof(S#state.groups),
                                   object_name()]}},
             {100,{call, ?MODULE, find_risk_object,
                                  [oneof(S#state.user_sessions),
                                   oneof(S#state.objects)]}},
             {20,{call, ?MODULE, update_risk_object,
                                  [oneof(S#state.user_sessions),
                                   oneof(S#state.objects)]}},
             {15,{call, ?MODULE, delete_risk_object,
                                  [oneof(S#state.user_sessions),
                                   oneof(S#state.objects)]}}])).
```

[5] Additional constraints, e.g., `user_sessions` not being empty for operations other than `login` to be eligible, have been omitted for the sake of clarity.

Apart from the definition of arguments generators, which can make use (as seen above[5]) of the values stored in the different fields of the internal state in each moment, the frequency distribution assigned to the different use cases included in this `command/1` function is part of the analytic effort required from the test designer, who will likely need a deep understanding of the application and business domain to be able to use them appropriately (either to faithfully represent a real system usage situation, or to favour certain operations over others in case some specific scenario wants to be stressed, etc.). Besides, as recommended in [137] and also applied in Section 6.3.1, the previous code uses symbolic calls instead of actual function calls in order to ease the debugging when an error is actually found.

For the sake of clarity, we will limit our following examples to just a few of the previous operations, namely `new_risk_group/2`, `find_risk_group/2`, and

delete_risk_group/2. Nevertheless, the procedure is repeatable for the rest of the use cases, as well as for the use cases in other system facades.

```
new_risk_group(SessionID, Name) ->
  {ok, GroupData} = risk_interface:new_risk_group(SessionID,
                                                   Name, ""),
  [{oid, GroupID}, _GroupCode, _Name, _Description] = GroupData,
  GroupID.

find_risk_group(SessionID, GroupID) ->
  {ok, _GroupData} = risk_interface:find_risk_group(SessionID,
                                                     GroupID),
  GroupID.

delete_risk_group(SessionID, GroupID) ->
  ok = risk_interface:remove_risk_group(SessionID, [GroupID]),
  GroupID.
```

As we can observe, the functions to be tested do not necessarily receive the same parameters we want to provide, nor do they return the same data. To cope with this contingency, these wrapper functions, local to the testing module in which the state machine is being specified, adapt the interface functions to the testing needs. The new_risk_group/2 function, for instance, serves as wrapper for the function new_risk_group/3 exported by the system facade risk_interface. In the original function, some parameters are of no use or interest for integration testing purposes, such as the group description; similarly, of all the information returned by that function, we are only interested in the unique identifier the system has assigned to the newly created object, since this is the kind of information we will store in the state machine internal state. Analogously, wrapper functions can be implemented for any state transition, this is, for any operation in command/1, avoiding to write calls to the real functions straight away. As we have already pointed out, in this way we can suit them to the best of testing interests without modifying the code we are actually checking.

Next, preconditions, postconditions, and next state functions need to be established for each state transition. In this case, ARMISTICE's client users can only perform the activities their roles grant them access to. Certain operations being restricted for certain user roles for security purposes is a common scenario in many systems. In particular, three different user profiles are defined in ARMISTICE: *administrator*, *manager*, and *clerk* (cf. Fig. 6.6). Only users with an *administrator* profile can insert, modify, or delete any business object, while *manager*s can do so for entities they are related to (i.e., bound to the same geographical location); *clerk*s cannot perform any administrative duties and are only granted read-access permissions to a subset of all business objects (again, the ones in their same location).

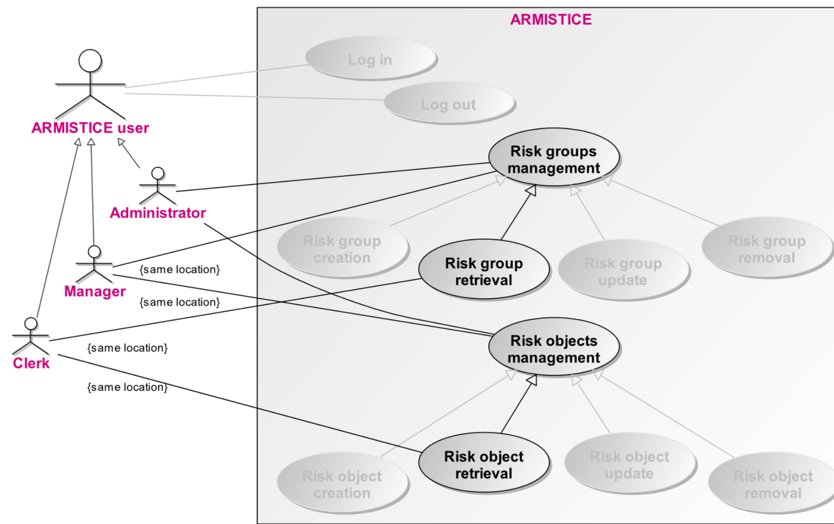


FIGURE 6.6. Role specialisation of selected management use cases.

Most user interfaces would, straight away, only show right options available for each user at each moment, as a reflection of such domain constraints. While this double-check of certain operational demands is not unusual, it should not lead us to make any groundless assumptions about user interactions. In other words, regardless of the user interface, the fact is that any of ARMISTICE's server public functions can be potentially invoked in any order, providing any given data as argument. Consequently, we should not rely on the component's caller (in this case, the user interface) if we do not want to add unrecognised bias to our integration tests. Since we want to test behaviour in all possible situations, our precondition functions will return `true` in every case:

```
precondition(_S, {call,?MODULE,
                         new_risk_group, [SessionID, Name]})->
  true;
precondition(_S, {call,?MODULE,
                         find_risk_group, [SessionID, GroupID]})->
  true;
precondition(_S, {call,?MODULE,
                         delete_risk_group, [SessionID, GroupID]})->
  true;
precondition(_State, _Call, _Arguments) ->
  true.
```

With regard to the effect of each function on the internal state (the lists of user sessions and objects created in the system), the `next_state/3` function modifies the internal state of the testing state machine according to the executed transition. For example, for the transitions of `new_risk_group/2` and `delete_risk_group/2` respectively, state change means adding the new

group identifier to the `groups` field (so that it can be used later in the testing sequence), or removing it (so that it is no longer used after that); on the other hand, the `find_risk_group/2` transition does not have any influence on the state, since it does not imply any changes:

```
next_state(State, Value,
            {call, ?MODULE, new_risk_group, _Arguments}) ->
    S#state{groups = [ Value | S#state.groups ]};
next_state(State, _V,
            {call, ?MODULE, find_risk_group, _Arguments}) ->
    S;
next_state(State, Value,
            {call, ?MODULE, delete_risk_group, _Arguments}) ->
    S#state{groups = lists:delete(Value, S#state.groups)};
next_state(State, _ReturnValue, _Call) ->
    State.
```

As for the postconditions, this is where we can be perform tests to check whether or not each operation result is what we should expect. In our application integration case study, we want to guarantee integration between clients and server more than the client or server internal behaviour (that needs to be, of course, checked on its own). So before we can go into detail with the postcondition functions we need to figure out how to check if ARMISTICE expected operations at the application back-end are actually invoked for each user interaction (i.e., use case invocation).

6.3.2.2 The dummy component strategy

As we know, when dealing with integration testing, the main goal is checking that two (or more) components work properly together. In other words, we want to be sure that when a service is requested from one component, which relies on another component to perform the operation, the former invokes the right methods from the latter.

A key reflection in this scenario is that, for such matter, the second component is not actually needed, because integration testing is essentially not about testing functionality, but about testing interaction. What we do need is a replacement that offers the same interface and provides the same kind of answers. We do not really care if the service is actually provided, if any operation at all is performed inside that *dummy* component, or if all the process of building up a request result is just simulated. Hence, as an integration testing strategy, we suggest creating and using such **dummy components** (also called *mock objects* [138]), which should be comparatively easy (they only need to conform to a given interface and simulate the same kind of responses), and includes advantages as faster response times and collateral effects avoidance (since services do not need to be actually provided, i.e., no databases need to be modified, no messages need to be transmitted, etc.).

In our case study, applying this dummy component strategy means that we can do without a full ARMISTICE server as long as we can replace its different parts with dummy components offering the same API and answers to the clients (so neither the real ARMISTICE server nor the existing clients need to be modified, cf. Fig. 6.7).

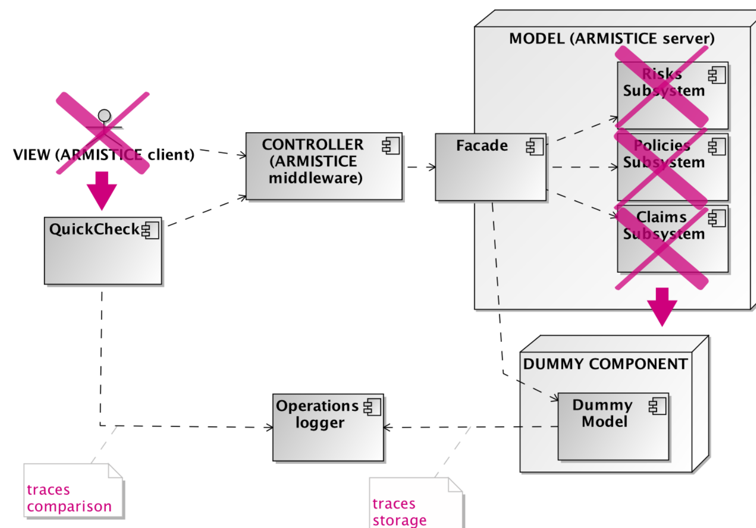


FIGURE 6.7. Usage of a dummy component for integration testing

The new dummy external component does not need to replicate the functionality of the original external component. It only has to provide the same interface and emulate its replies. For the use cases we have been using as example, a dummy component will not actually insert, retrieve, or delete any risk group or risk object, but from a external point of view it will seem exactly as if it did, because it will provide all those operations supported by the original component (or at least, those under test) and return similar responses to the invoker. Thus, the difference between the original component and the new dummy component will not be noticeable to the observer.

Yet, there is something more we need to do to be really capable of checking the proper connection between component requests and called functions, which is precisely the key aspect to check on an integration test. To be able to inspect whether the client's queries are forwarded to the expected exported services, the dummy component shall register all access to its interface functions, and also provide a way to retrieve that information. These calling traces can later be recovered from our QuickCheck testing module to verify that the correct interactions are taking place in all possible scenarios.

For our integration testing example, we implemented a simple and independent *generic server* (using the Erlang `gen_server` behaviour [107]), where the dummy components can register function invocation data. This module, called `operations_logger` (cf. Fig. 6.7), provides the following interface:

- `add_operation/1`: adds an operation `Operation` to the list of invoked operations, stored in the generic server state,
- `get_all_operations/0`: returns the list of invoked operations, stored since last call in the `gen_server` state, and clears that list;

where each operation `Operation` is a tuple with elements:

```
{Module, Function, Arguments, Result}
```

Every time a function in the dummy external component is called, it is registered via the `operations_logger` module. In our study case, thus, this is how the original `new_risk_group/3` function implementation, which is one of the functions of the ARMISTICE server API (exported by the `risk_interface` facade), looks like:

```
new_risk_group(SessionID, GroupName, GroupDescription)->
  {ok, GroupData} = risk_group_facade:new_skeleton(SessionID),
  [{oid, GroupID}, {code, GroupCode}, _, _] = GroupData,
  ok = risk_group_facade:update(SessionID, GroupID,
                                GroupName, GroupDescription),
  ok = risk_group_facade:unlock(SessionID, GroupID),
  {ok, [{oid, GroupID}, {code, GroupCode},
        {name, GroupName}, {desc, GroupDescription}]}.
```

According to this, the sequence of steps for the creation of a risk group in the system involves first creating a skeleton of the object, which automatically assigns an object identifier and an entity code. Then, the object is updated with specific values for name and description, and it is finally released for public access to other ARMISTICE users. These functions (`new_skeleton`, `update`, or `unlock`) belong to more internal subsystem facades (in this particular case, to the `risk_group_facade`). Respectively, below is how the same function will look like in our dummy component:

```
new_risk_group(SessionID, GroupName, GroupDescription)->
  GroupData = [{oid, group_id()}, {code, group_code()},
               {name, GroupName}, {desc, GroupDescription}],
  operations_logger:add_operation({risk_group_facade, new_skeleton,
                                   [SessionID], {ok, GroupData}}),
  operations_logger:add_operation({risk_group_facade, update,
                                   [SessionID, GroupID,
                                    GroupCode, GroupName,
                                    GroupDescription], ok}),
  operations_logger:add_operation({risk_group_facade, unlock,
                                   [SessionID, GroupID], ok}),
  {ok, GroupData}.
```

were we have basically substituted function calls for invocation registrations on the log server, and emulated the same kind of answer by taking action only in specific sensible data (such as risk group identifier) which will be of use for testing purposes.

On the basis of the previous example, it is easy to realise the high potential for automatisation that the process of dummy components creation has. Just by inspecting the facade(s) under test, we could automatically generate a dummy version of them by replacing function calls by logging operations of those method invocations. Alternatively, communications between facades could also be monitored in a way that those invocations could be detected, logged, and made accessible for interaction pattern verification.

The same procedure is to be used in all the functions of the dummy component, so that for each operation to test at the QuickCheck module, the steps that are needed to handle each service request are registered. Then, the way of checking which functions were called requires a slight modification to the implementation of the wrapper functions we saw before. Additionally, they need to retrieve (from the log server) the set of function calls that the emulated execution of the service implied, and return them as part of the wrapper customised result:

```
new_risk_group(SessionID, Name)->
  {ok, GroupData} =
             dummy_risk_interface:new_risk_group(SessionID,
                                                      Name, ""),
  [{oid, GroupID}, _GroupCode, _Name, _Description] = GroupData,
  Operations = operations_logger:get_all_operations(),
  {GroupID, Operations}.
```

For the `new_risk_group/2` example above, after calling the dummy version of the function under test, the set of operations invoked as a consequence is retrieved from the auxiliary module that stores them (`operations_logger`). As has already been said, this set of operations should be inspected to check that the proper calls are being made for this operation, but the actual test is not made as part of the wrapper implementation; instead, it represents exactly the postcondition that the operation needs to satisfy in order to be considered correct. Thus, in order to make available for the postcondition the `Operations` information, the wrapper function should return it as part of its result. Remember that `Operations` include exhaustive information about calls (function name, arguments, result), so should an operation depend on the results of previous operations, or on the values provided as function arguments, all that information would be available.

The postcondition for the `new_risk_group/2` function will check the list of invoked functions to be sure that all (and only) the appropriate dummy ARMISTICE model functions have been ‘executed’:

```
postcondition(State,{call,?MODULE,new_risk_group,Arguments},R)->
  {Result, Operations} = R,
  check(new_risk_group, {State, Arguments, Operations, Result});
```

In this source code extract, it is actually the `check/2` auxiliary function which has the knowledge about the correct list of operations to be called to create a new risk group, and compares it with the list of operations obtained from `operations_logger` (`Operations`). The boolean result of that comparison is the answer to the operation postcondition, and similarly for the rest of the use cases to be tested.

A sequence of operations randomly generated by QuickCheck could be:

```
{call,armistice_eqc,login,["cvazquez"]}
{call,armistice_eqc,login,["fpoza"]}
{call,armistice_eqc,new_risk_group,["cvazquez","vehicles"]}
{call,armistice_eqc,new_risk_group,"["fpoza","warehouses"]}
{call,armistice_eqc,login,["mfernandez"]}
{call,armistice_eqc,new_risk_group,["fpoza","shops"]}
{call,armistice_eqc,find_risk_group,["mfernandez", 10]}
{call,armistice_eqc,logout,["fpoza"]}
{call,armistice_eqc,find_risk_group,["cvazquez", 20]}
{call,armistice_eqc,delete_risk_group,["cvazquez", 20]}
```

for which the corresponding internal invocations trace, stored at the auxiliary log module after the execution of such sample test, case would be:

```
{users_facade,login,["cvazquez"],"cvazquez"}
{users_facade,login,["fpoza"],"fpoza"}
{risk_group_facade,new_skeleton,["cvazquez"],{ok,[{oid,10},
                                 {code,"G1"},{name,""},{desc,""}]]}
{risk_group_facade,update,["cvazquez",10,"G1","vehicles",""],ok}
{risk_group_facade,unlock,["cvazquez",10],ok}
{risk_group_facade,new_skeleton,["fpoza"],{ok,[{oid,20},
                                 {code,"G2"},{name,""},{desc,""}]]}
{risk_group_facade,update,["fpoza",20,"G2","warehouses",""],ok}
{risk_group_facade,unlock,["fpoza",20],ok}
{users_facade,login,["mfernandez"],"mfernandez"}
{risk_group_facade,new_skeleton,["fpoza"],{ok,[{oid,30},
                                 {code,"G3"},{name,""},{desc,""}]]}
{risk_group_facade,update,["fpoza",30,"G3","shops",""],ok}
{risk_group_facade,unlock,["fpoza",30],ok}
{risk_group_facade,find_risk_group,["mfernandez", 10]}
{users_facade,logout,["fpoza"],ok}
{risk_group_facade,find_risk_group,["cvazquez", 20]}
{risk_group_facade,findlock_risk_group,["cvazquez", 20]}
{risk_group_facade,delete_risk_group,["cvazquez", 20]}
```

We can see that, complex services invocation such as the creation of a new risk group translates into several internal model function calls (`new_skeleton`, `update`, `unlock`), as expected.

On real invocation, and according to our definitions, QuickCheck generates and executes several iterations of tests where each test case is itself a randomly long sequence of the specified operations. Thanks to the randomness introduced by the tool, perfectly common function call combinations (as in the typical ones a user/client of the system will demand) will be tested together with more unusual or even completely improbable sequences, thus improving the chances of finding an error.

As we already saw in Section 6.3.1, whenever QuickCheck finds a sequence of operations that leads either to a crash of the system or to a false postcondition, the test execution is stopped and the complete trace of operations is reported. QuickCheck performs also a *shrinking* stage on the faulty trace to find its smallest equivalent, that produces the same wrong behaviour. However, in this case, after executing a great number of tests, no errors were found. Hence, even though we cannot prove that the ARMISTICE subsystems integration is correct, we have obtained great confidence in the fact that for each operation in the ARMISTICE server public API, the correct operations in the internal business logic are called, and in the correct order. Of course, since we are testing against a dummy component, this fact does not imply that the entire system works properly, and individual component testing should always be done separately.

6.3.2.3 Negative testing

The testing procedure we have just explained describes a convenient way to perform integration testing that can easily be applied in similar scenarios. The key aspect of the proposed approach is the implementation of dummy components which offer the same interface and emulate the same behaviour as the original components, in order to avoid any changes in the component to test.

However, nowadays it is hardly ever the case that all components in a software system are deployed in an isolated and unique environment. In such circumstances, there is a whole range of potential failures, due not only to the failure of a component, but also to external communication conditions. When testing interaction between components, relevant questions such as what happens if one of them fails to respond to a request need to be taken into account. Such failure scenarios fall outside of the scope of the state-machine integration testing that we have described so far. The kind of testing we have suggested with this state-machine based methodology only focuses on *positive testing* right now, but situations in which function invocations may fail due to external circumstances (such as network congestion or similar) must also be considered. If they are not, then we will never be able to test how such errors affect

the behaviour of our system and control whether these unavoidable errors are handled in a satisfactory manner.

To deal with **negative integration testing**, again we would prefer not to modify the original components, therefore we must re-use the dummy components for simulating these situations too. For instance, a possible way to reproduce timeouts is forcing the dummy component not to reply to requests immediately, but wait several seconds before returning the result value, or alternatively to drop the normal answer. By doing so, new communication scenarios are added, which obviously need to be properly handled at the corresponding postconditions. If test sequences with these new use cases are generated and executed, i.e., including waiting times and exceptions in the communication between components, tests may fail because the expected call traces will be different from those obtained when such exceptions or errors were not considered. So, a key aspect here is to properly deal with these anomalies in the postcondition function, where we can act accordingly, inspecting the traces taking into account these irregular situations.

Since behaviour might be different depending on the situation, knowledge about whether the dummy component will enforce a delay or not, is needed in the QuickCheck testing module. Thus, a good solution is to implement a new transition in the QuickCheck state machine to anticipate a delayed interaction. In combination with the `frequency/0` utility function, the state of the network can be simulated for different congestion scenarios:

```
command(S) ->
  frequency([{1, {call,?MODULE, delay,
                        [?LET(MSec,choose(1,10000),(10000-MSec))]}},
             {10, oneof([{call, ?MODULE, use_case1, [data_gen1()]},
                         {call, ?MODULE, use_case2, [data_gen2()]},
                         ...])},
             ...]).
```

The sample code above implies that once in ten times, a delay will be forced. The `delay` transition, uses the `choose/1` QuickCheck generator, which generates a random number in the specified range (in this case, to produce a delay between 1 and 10000 milliseconds).

The implementation of the `delay/1` function in the testing module must communicate with the dummy component to set the amount of time the next response shall be delayed. We should not ignore the fact that we are actually introducing a communication pattern to the original scenario, but it only takes place between the QuickCheck testing module and the dummy component. A new function, `delay/2` is implemented in the dummy component, while the original component remains untouched, and each dummy operation will now take into account the notified delay prior to returning the responses.

Besides, the structure of the QuickCheck state machine would also need to be modified, since the response time of the next request must be handled. Thus, a new field `response_time` is added to the state:

```
-record(state, {user_sessions, groups, objects, response_time}).
```

This new field will be filled in the `next_state/3` function for the new transition `delay`,

```
next_state(State, _Result, {call, ?MODULE, delay, [MSec]})->
  State#state{response_time = MSec};
```

In this way, before executing the next function invocation in the test sequence, whichever it is, the response time will be accessible. After execution of the chosen function, the response time should be reset (at the corresponding `next_state/3` transition):

```
next_state(State, Result, {call, ?MODULE, use_case1, Arguments})->
  ...,
  State#state{response_time = 0};
next_state(State, Result, {call, ?MODULE, use_case2, Arguments})->
  ...,
  State#state{response_time = 0};
...
```

With respect to the `precondition` function, we may want to stipulate that the `delay` function is not called twice consecutively, i.e., if a delay already exists, another delay would not overwrite it:

```
precondition(State, {call, ?MODULE, delay, [_MSec]})->
  State#state.response_time == 0;
```

The `postcondition` function will always return `true` for this command, and will be modified according to expected behaviour (if needed) for the rest of commands.

With this approach, the delay is in a way a global parameter which is applied regardless of the operation chosen to be executed next in the testing sequence. However, more precise situations can be configured, for instance defining a specific delay for each particular function of the dummy component. For such a testing scenario, we would have one new delayed transition for each original function in the QuickCheck state machine, with the delay being an additional parameter to the wrapper function and its dummy counterpart. In that way, very different and changing testing scenarios could be reproduced.

6.3.3 Business rules validation

Up to this point, we have explored different software testing scenarios at distinct application levels, from the lowest and most elementary (unit testing) to more complex situations (integration testing). Also, testing strategies for them have been designed, developed and exercised as complete approaches which help obtaining trustable results in each of those cases. Now, as our last experience in this area, we face the highest abstraction level of software validation: that of business concepts and domain rules.

Nowadays, many applications and information systems are data-intensive, and most of them use a database to handle vast amounts of data, relying on trustworthy data management systems for data storage. Modern database management systems (DBMS) provide transactionality and fault-tolerance features (e.g., ACID properties [139]: atomicity, consistency, isolation, durability), ensuring the data they accumulate and manipulate survives even if an external application using that data fails.

The most commonly used DBMSs are Relational DBMSs. RDBMSs have been widely used for the last thirty years and, besides storing information in a structured manner [140], they also allow the definition of several types of constraints on the data, to keep it free of basic inconsistencies. Relationships on data constrain the data itself; thus the database content, and those constraints are guaranteed to be satisfied by the RDBMS. In other words, storing data in a relational database can only succeed if the relational constraints are respected. Also, the combination of pre-production testing and massive in-use testing by both industrial and commercial clients, provides enough confidence in that no mature DBMS loses data during normal operation and constraints are always consistently satisfied.

However, it is not always convenient, or even possible, to place all responsibility concerning consistency checks onto the DBMS. Some data constraints can be extremely complicated. They may involve not only data formats or relationships, but also non-trivial calculations; their applicability may be time-dependent; or their complexity so high that performance and efficiency can become an issue. Commonly, the constraints are also very business-specific, and therefore there is no need, nor desire, to "hard-wire" them in the database (particularly if the database is, or will be, shared by other systems or applications). For these reasons, moving the consistency checks for these business-specific constraints, also called **business rules** [141] to the *business logic* layer, on top of the persistence management layer and the DBMS (cf. Fig. 6.8), is a better design alternative. If the application respects its own layered structure to access the data, then the data in the database should be consistent with the business rules implemented by the business logic at all times.

So we find here a common scenario for data-intensive systems, where business domain modelling requires the enforcement of specific and complex busi-

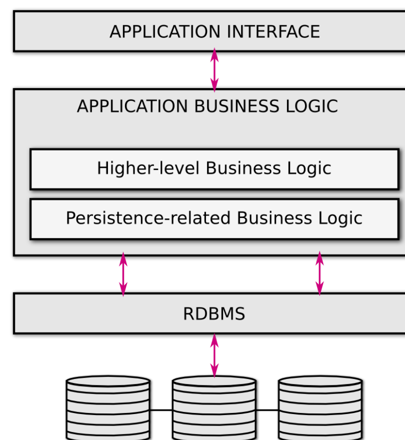


FIGURE 6.8. Layered application architecture sample

ness constraints. Unfortunately, none of the testing approaches we have applied so far fits this new testing demand, where the most relevant aspect is to provide enough confidence in that no situation can be reached where a business rule is violated and data is potentially left in an inconsistent state. No matter if that is a consequence of one or several invocations of system services, or a result of unexpected input, faulty internal behaviour, or even unconsidered scenarios.

Literature shows that previous attempts have faced the challenge of business rules *definition and modelling*, either by proposing new development methodologies, or by formulating modifications to existing ones [141, 142]. Nonetheless, few efforts have been devoted to *testing* the actual fulfilment of such business rules. To the best of our knowledge, the most practical approach to this matter is the one presented in [143], which proposes an extension to JUnit [144] to perform better testing, namely data-oriented tests, for business rules testing.

In this last section, we present a different approach to business rules validation. Instead of manually adding conditions to an existing test suite, tests are automatically generated (and executed) from a model using QuickCheck, covering both positive and negative testing.

6.3.3.1 Formulating business rules

Before elaborating on the general method for testing data intensive applications we have developed, we introduce a simple case that serves as motivating example. Our strategy is used for testing that a properly architectured software system does not violate its business rules when data access is performed through the business logic layer. We describe our approach on the basis of a small, but representative, application. The technique generalises to

much larger systems, and we applied it to our ARMISTICE case study. After being developed for ARMISTICE, the same technique has been successfully evaluated in a different industrial setting, demonstrating its general applicability [145].

Domain-specific business rules play a major role in the well-functioning of any data-intensive system. However, while data-intrinsic constraints are clearly identified, located at database level, and defined and modelled using ER diagrams and other data model artefacts, that is not regularly the case for business level constraints. Some authors suggest that, similar to what is done in knowledge-based systems, where the business rules (inference rules) are treated in an explicit way and placed in the same location (rule base), the same should be done in any information system [146]. This will, of course, provide a unique repository for business rules, to which all other parts of the system will need to be granted access in order to implement the business logic, and thus the system services and functionalities. However, this idea, inherited from the AI field, implies greater system complexity, along with other problems such as more expensive maintenance operations and poorer efficiency [146]. The usual and more common scenario is, in practise, that of software applications in which business rules, whilst possibly written down as requirements or specifications in the system documentation, are spread across the system, i.e., not explicitly, but implicitly coded in the implementation of the different services, methods, or functions.

In any case, when testing this kind of software, it does not make a big difference whether the business rules are implicit or explicit within the system. The testing efforts must always be devoted to pursue evidence that they are either executed or else somehow implied by the working system. At the testing stage, this can be achieved in an implementation-independent and non-intrusive manner by looking at the effects (i.e., results, output, database changes) of system operations.

To illustrate this, we borrow a simple example from Willmor and Embury [143], and we implement a database application to deal with customers, products, and orders, in which we introduce a *status* for customers, either ‘gold’ or ‘non-gold’. The purpose of the status is reflected in that only gold members can purchase some special products. An ER diagram for this example is shown in Figure 6.9.

When translating Diagram 6.9 into a relational database schema, a set of constraints appear, that will be implemented as database constraints:

- Entity keys (i.e., unique identifiers such as customer *ID*, order *number*, product *code*) will be translated into primary key constraints.
- One-to-many relationships will be translated into foreign key constraints (i.e., *customer ID* on relation *order*).

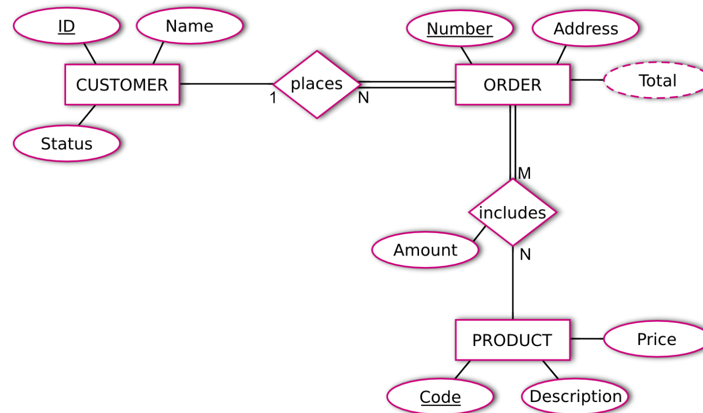


FIGURE 6.9. ER diagram example

- Many-to-many relationships will be translated into relations with two foreign key constraints (i.e., *order number* and *product code* on new relation *order products*).
- Other flavours of constraints may be added (i.e., non null product *price* or *amount*).

However, constraints like ***when*** a customer acquires 'gold' status, or ***which products*** are only to be purchased by 'gold' customers, are the kind of constraints not desirable to implement at the database level. Not only because these constraints may vary during system life or operation (the list of 'featured products' can be very dynamic, a 'gold' membership can be obtained by purchasing a number of orders or just be granted temporarily and selectively to incentivise consumer shopping), but most importantly, it is a domain-level property which is not data-intrinsic. Thus, these constraints are considered business rules and are taken care of at the application business level.

As stated in [143], correct implementation of business rules by a system can be tested by translating each rule into some form of SQL query [147] which can be evaluated against the database. For instance, the following SQL sentence will check that customers without 'gold' status cannot have an order placed that includes 'gold' featured products. The business rule is violated if anything else than the empty set is returned.

Business rule:

```
SELECT customer.id
  FROM customer, order, order_products
 WHERE customer.id  = order.customer_id
   AND order.number = order_products.order_number
```

```
  AND customer.status <> 'gold'
  AND order_products.product_code
                          IN <featured product list>
```

Assuming at this point that gold membership is obtained when the user has at least five orders placed, and if the available operations in the system allow to add, update, and delete entities (customers, products, orders), there are many ways in which the previous business rule can be broken. Not only single operations like placing a new order including a featured product when the user does not have the gold status will violate the constraint above, the following sequence will do as well:

1. A non-gold customer places the fifth order (the system, apart from storing the order information, updates customer status to 'gold').
2. The same customer (now upgraded to 'gold') places an order for a featured product.
3. The same customer cancels one of the first five orders.

There are different valid system behaviours that can be implemented to avoid reaching the inconsistent situation that this sequence will lead the database to (according to business rules). The application can prevent a gold customer from cancelling an order if that would remove her gold status, or else cancelling such order could not only remove the gold membership condition, but also automatically cancel the order that included the featured product, for instance. When testing database applications for data consistency, we do not really care about which policy is actually implemented: we aim to ensure that, whichever it is, it guarantees business rule compliance at all times.

From the previous example we conclude also that data-intensive application tests must include sequences of system services invocation. Straight violations of business rules will usually be taken care of by the developers during unit testing, but uncommon or atypical sequences of invocations (especially to different system components or subsystem functionalities) may very well cause inconsistencies on data, and are more likely to be overlooked. In real applications, though, the amount of possible sequences of interface function calls is enormous. In particular, it is impractical to generate a large number of such sequences manually.

Besides, even though user interfaces can disable certain sequences of activities, such as preventing a non-gold user from ordering a featured product by not displaying that option, we cannot guarantee nor trust them to do so, hence we want to generate this kind of testing sequences as well. We consider sequences that we expect the application to produce during normal operation *positive tests*, and those that we expect the application not to produce *negative tests*.

To help generating such sequences, and also to get around unconscious presumptions that would lead to the exclusion of relevant scenarios or possibilities, automatic test generation tools that provide random input can be helpful. However, completely randomising a test sequence would generate lots of meaningless tests, e.g., repeatedly trying to add non-existing products to an order, or cancelling unknown orders, etc. In pure random generation, meaningful test sequences, like the previous one involving the same customer performing three different operations, will hardly ever occur unless the test generation is steered to some extent. As a suitable way of facing all these challenges, we chose to apply QuickCheck state-machine testing once more.

6.3.3.2 Applying a state machine model

The business logic of our shop example contains a set of typically exported functions to register a new customer (`new_customer`) or to disable a certain product for sale (`delete_product`), among more complex services such as registering that a certain customer has placed an order (`place_order`). The safest assumption for testing is to presume that any interface function defined in the business logic can be called in arbitrary sequence with arbitrary arguments. But, as we previously mentioned, we want to get as many meaningful tests as possible; therefore we want to have (and use) knowledge about which interface calls have been previously invoked and the results they had, in order to generate the next interface call in a testing sequence. For example, the failing scenario described on page 133 would be the result of a sequence of interface calls like this:

```
Id    = {call, business_logic, new_customer, ["Laura"]},
Nr1   = {call, business_logic, place_order,  [Id,1277]},
Nr2   = {call, business_logic, place_order,  [Id,7027]},
Nr3   = {call, business_logic, place_order,  [Id,3112]},
Nr4   = {call, business_logic, place_order,  [Id,4983]},
Gold1 = {call, business_logic, place_order,  [Id,9002]},
        {call, business_logic, cancel_order, [Nr3]     }
```

where we need a return value from the first call in consecutive calls, and we need to remember the returned order numbers in order to cancel one.

The core of a data-intensive system is, ultimately, the database behind it, thus it can be seen, as a whole, as a stateful system where the ‘state’ is actually all the data stored in the database at any particular moment. The way the database is brought from one state into another is through the different system services available, which we can specify as state machine transitions.

The challenge here is to abstract from the data in the database and use that abstraction as a model for the state machine. Otherwise, we would end up with a copy of the entire database as state. Not only would that state be potentially too large for a model, more seriously, the entire business logic would

have to be re-implemented to perform the state transitions, since re-using the implementation under test would not enable us to find errors. Of course, implementing software twice is an unattractive idea; apart from the work involved, there is the possibility of making similar mistakes.

Therefore, to automatically generate meaningful both positive and negative tests and keep a minimum test state at the same time, we need to make sure we maintain the least amount of data needed to generate related interface calls, and enough data to bring the system in all kind of different states.

Since state changes are a consequence of public interface function invocations, the list of possible function invocations constitutes the state machine set of transitions or *commands*, as explained in Section 6.3.2. To specify them, the QuickCheck's `command/1` generator is used, conveniently in combination with the `frequency/1` function to assign to each transition a certain likelihood of appearance in the generated testing sequences (according to domain knowledge):

```
command(S) ->
  frequency(
    [{1, {call, business_logic, new_customer,[name()]}},
     ...
     {4, {call, business_logic, add_product, [...]}},
     {2, {call, business_logic, place_order, [...]}}
     ...]).
```

The `command/1` generator takes the test machine abstract state as input, and commands are generated relative to the present abstract state. As input for each of the specified interface functions, specific generators need to be defined as well. In our simplistic example, for instance, it is easy to specify a generator for the argument of `new_customer` because only a name is required, but even if address, email address, telephone number, etc., needed to be provided, random data could be generated for these fields resembling the SQL definitions from the application database schema. For instance, the next database table description:

```
CREATE TABLE customer
  (id INTEGER CONSTRAINT cust_prk PRIMARY KEY,
   name VARCHAR,
   status VARCHAR CONSTRAINT cust_nn NOT NULL)
```

can be translated into the set of QuickCheck generators:

```
id() ->
  int().
```

```
name() ->
  varchar().

status ->
  oneof(["gold", "non-gold"]).
```

Generators can be as specific as needed, e.g., creating the name as a first name and a last name both starting with an uppercase character and the rest in lowercase. That, however, would only be significant in case such formatting is enforced by the application interface functions. If that is not the case, the recommendation is to remain as general an simple as possible. Besides, in this case the business logic creates new customers always with `non-gold` status, and assigns an automatically generated unique ID, so both `id/0` and `status/0` generators are not necessary.

Understandably, generating only random parameters for all functions means that services such as `place_order` would most likely be called with customer IDs and product codes that are not in the database. In the intentional testing approach [143] objects are automatically generated when not present in the database. We propose a different approach and use the abstract state (`S`) to keep track of created customers and products in order to use them in calls to `place_order` and similar. In addition, we still choose now and then, with a small probability, a customer or product (or both, i.e., double error) that does not exist in the database to validate the generation of adequate error messages (negative testing).

```
command(S) ->
  frequency(
    [{1, {call, business_logic, new_customer, [name()]}}, ...
     {2, {call, business_logic, place_order,
                                [customer_id(S), product_code(S)]}}]).

customer_id(State) ->
  oneof([int()|keys(State#shop.customers)]).

product_code(State) ->
  oneof([int()|State#shop.products]).
```

Generators `customer_id` and `product_code` in the previous source code pick a random integer (`int/0`) and add it to the list of known keys (function `keys/1` extracts the keys from the customer records); the QuickCheck primitive `oneof/1` takes then a random element from that list, hence generating both positive and negative (with a lower probability) testing sequences.

The abstract state of the state machine is defined to be a record data structure with two fields, each one containing a list which will be initially empty: customer references, and product codes.

```
initial_state() ->
  #shop{customers=[],
        products =[]}.
```

The customer reference list will contain a list of `customer` records, i.e., two-field tuples storing the customer identifier (`id`) and the list of orders placed by the customer,

```
  #customer{id, orders=[]}
```

where an order is stored by its order identifier, needed for cancelling operations, for instance. These lists will be populated, during test generation and execution, by customer and product data created and added to the database as testing proceeds, and potentially re-used in further test cases.

Each interface function may have an influence on the state, and the state transition callback function `next_state` is filled to reflect that. The `next_state` function returns the updated abstract state, i.e., the abstract state we expect the database to be in after executing the interface call.

```
next_state(State, Id, {call, ?MODULE, new_customer, [Name]}) ->
  NewCustomer = #customer{id = Id},
  State#shop{customers = [NewCustomer | State#shop.customers]};
...
next_state(State, Nr,
              {call, ?MODULE, place_order, [Id, ProductCode]}) ->
  case existing(ProductCode, State#shop.products) of
    true ->
      case get_customer(State#shop.customers,Id) of
        not_found ->
          State;
        Customer ->
          NewOrders = [Nr | Customer#customer.orders],
          State#shop{customers =
            [Customer#customer{order = NewOrders}
                          | delete_customer(S#shop.customers,Id)]}
      end;
    false ->
      State
  end;
...
```

In this example, the first function clause specifies that when a new customer is created, the field `id` of a new `customer` record is initiated with the value that the `new_customer` function will return when executed (namely, the unique identifier assigned to the new object, `Id`)[6]. After that, the new customer is added to the list of already existing customers (`State#shop.customer`) and

[6] The other field, `products`, gets a default value, the empty list.

`State` is updated by replacing the value of the field `customers` by this new list of customers. The second function clause establishes a different state according to the existence of the user and product in a `place_order` operation. No changes are produced if any of them does not exist (`State` is returned unchanged); should customer and product be already present in the abstract state, it would mean that they have been produced by other transactions earlier in the test case, so the order can be potentially placed, and the order identifier returned (`Nr`) added to the existing orders of the customer.

Now these `initial_state` and `next_state` functions allow QuickCheck to generate a great amount of different commands that are partially random and partially refer to expected previously returned values.

As seen in Section 6.2.2, two more callback functions need to be defined before we can actually start testing our application with this state machine QuickCheck library: `precondition/2` and `postcondition/3`. However, unlike other scenarios, data-intensive testing will not make use of those library hooks. First of all, because we do not want to impose any restriction on which transitions are produced after each other, so no previous condition would be asked in any case for an exported function to be invoked; secondly, because postconditions cannot be effectively used in this business rules testing scenario, so function result verifications need to be performed in a different place.

Assuming that the business rules hold in our initial state, after execution of each test case, we would like to verify that they still do. The natural location for business constraints checking (formulated as SQL-queries, just as we saw before) would be as part of the postconditions, but once a command has been executed, the database is already changed, so there is no possible way of figuring out whether the new database state is consistent with the business rules and the *original* database state, unless we include in the QuickCheck state machine internal state as much information as needed to reproduce the previous database content. Similarly, if a service or function invocation returns an error, the minimum information stored at the QuickCheck state machine internal state will usually not be enough to check if that error was expected according to database original state and the details of the invocation call. Of course, increasing the amount of data in the internal state could help, but it would lead us closer and closer to replicating the database content, which is something we want to avoid.

Similarly, checking the business rules cannot be done either as part of the `next_state/3` function, because of symbolic values usage. Even though working with symbolic calls has a number of advantages, it also has a drawback: since we do not work with evaluated values, we cannot observe whether a certain action will be successful or not when this would depend on the actual values of the invocation, thus we cannot predict the final situation such a test case will take the system to in terms of the database content.

In the absence of a full internal state that would replicate database contents, and unable to perform the desired checks either at the `next_state/3` nor at the `postcondition/3` functions, we propose as alternative predicting the expected result in advance querying the actual database state (prior to functionality execution), and contrasting it with the result obtained from the system after executing the test.

6.3.3.3 Results validation

When run, these data-intensive test cases would fail if the application crashes during consecutive calls, when the result returned by a function is unexpected, or when the database is left in a state that violates the business rules. To check that each interface function returns an expected value (keeping in mind that we want to keep the test state as simple as possible), we implement a **validation function** for each transition in the state machine definition, which will basically describe (and test) the conditions on the database that should be fulfilled in order to create a certain return value. This is easiest done by replacing the calls to the interface functions by local versions (wrappers) which combine validation and actual interface call.

In our shop example, for instance, it is always possible to create a new customer and get a new identifier in return. We cannot possibly know which identifier will be returned, but there is no need for us to be aware of the precise value. Therefore, our local version of the `new_customer` function just invokes the corresponding interface function and matches the *kind* of result:

```
new_customer(Name) ->
  Result = business_logic:new_customer(Name),
  case Result of
    Id when is_integer(Id) -> Result;
    _Other                 -> exit(unexpected_value)
  end.
```

In this particular case (`new_customer/1`), no conditions on the database need to be inspected. If the returned value fits the expected *type of* value (i.e., is an integer), then the value generated by the business logic is returned; otherwise, an exception is raised and the test fails.

More advanced interface functions such as `place_order`, however, demand a more complex validator function. A specific user placing an order for a particular product represents a state change that can potentially violate a business rule:

Business rule:

> "*If an order contains a featured product,*
> *the customer must be a gold customer.*"

Intuitively, we infer from this description that if a customer has 'gold' status, placing an order should always be a successful operation (provided that both the user and the product actually exist in the database), and if the customer does not have the required status, the interface call will only be allowed if the order does not contain a featured product. These data-related conditions are easily translated into boolean functions with specific SQL-queries:

```
product_exists(Connection, Code) ->
  [] =/= db:process_query(Connection,
      "SELECT code FROM product WHERE code="
        ++ integer_to_list(Code)).

customer_exists(Connection, Id) ->
  [] =/= db:process_query(Connection,
      "SELECT id FROM customer WHERE id="
        ++ integer_to_list(Id)).

featured_product(Connection, Code) ->
  [Code] == db:process_query(Connection,
      "SELECT code FROM product WHERE code="
        ++ integer_to_list(Code) ++
        " AND code IN " ++ featured_products()).

gold_customer(Connection, Id) ->
  [Id] == db:process_query(Connection,
      "SELECT id FROM customer WHERE id="
        ++ integer_to_list(Id) ++ " AND status='gold'").
```

Some of the above queries may well be equal to existing functions in the business logic layer. Yet choosing not to re-use them has the value of separating concerns and specifying the test constraints in the test specification. In particular, if different people write tests specification and business logic layer, there is an extra possibility to identify failures.

Now we evaluate all involved constraints before executing the interface function, and check whether the result from the business logic invocation is justified according to the previous database state:

```
place_order(Id, ProdCode) ->
  PE = product_exists(ProdCode),
  CE = customer_exists(Id),
  FP = feature_product(ProdCode),
  GC = gold_customer(Id),
  Result = business_logic:place_order(Id, ProdCode),
  case Result of
    OrderId when is_integer(OrderId) and
                 PE and CE and (not FP orelse GC) -> Result;
    {error,not_gold_customer} when FP and not GC  -> Result;
    {error,non_existing_product} when not PE      -> Result;
    {error,non_existing_customer} when not CE     -> Result;
```

```
    _Other                                     -> exit(unexpected_value)
  end.
```

We implement a case distinction on the obtained result instead of on the combination of previous conditions. On the one hand, this reduces the amount of code, since only one alternative for each possible return value is needed. On the other hand, we specify different from what we would do when implementing the business logic layer. If the code is different, it is less likely that the same mistake is made. But even more important than the two reasons above, we do not have to worry about the (normally) unspecified implementations of dealing with double faults; in case both product and customer do not exist, there might not be information on which of the errors is produced. By looking at the result, no matter which of the two error messages is produced, we accept the error message.

Two additional remarks deserve attention here, the first being that we always have to inspect the database state prior to computing the interface call, since in case the latter results in a state change, the prediction would be useless if performed afterwards. Secondly, all local versions of interface functions that wrap together database state querying and service invocation should, for security and integrity purposes, be embedded in a single database transaction, making each procedure atomic and therefore avoiding problems if testing in a concurrent environment. Unfortunately, not all DBMSs support nested transactions, which makes it essential to perform these tests sequentially, with just one client querying the database. In most cases, all interface calls would be embedded in their own transaction, so this sequential testing would be no real obstacle. However, there may be issues in a concurrent setting that could not be discovered with this testing approach, and for which other testing techniques (such as load testing, for instance) would be of use.

As far as the integrity of the business data is concerned, checking business rules consistence after executing each command separately is no different from doing so at the end of the whole testing sequence. Our assumption is that if the database is brought in a state that is inconsistent with the set of business rules, then the execution of consecutive commands will be unlikely to repair it. Moreover, if we run thousands of test sequences and the business logic can be violated with a sequence that repairs the violation afterwards, then we most likely also run, as one of the other tests, a shorter sequence that violates the logic and does not have consecutive commands which amend the situation. The advantage of checking violation of the business logic only once per test sequence is that it makes testing faster, and hence more tests can be run in the same time.

In order to test compliance of the database after-test status with the business logic constraints, we add a check for each business rule and perform all these as part of an invariant function after executing the generated sequence. Each

business rule is embedded in a database transaction consisting of one or more SQL queries. For example, the invariant for our running example is:

```
invariant() ->
  {ok, Connection} = db_interface:start_transaction(),
  Result = business_rule(Connection),
  db_interface:rollback_transaction(Connection),
  Result.

business_rule(Connection) ->
  [] == db_interface:process_query(Connection,
      "SELECT id "
      "  FROM order_products NATURAL JOIN customer "
      " WHERE status <> 'gold' "
      "   AND code IN ( " ++
              string:join([integer_to_list(P)
                     || P <- ?GOLDEN_PRODUCTS], "," ) ++
      " )").
```

Business-rules compliance is then inspected before and after a whole test sequence has been executed, defining the main testing property as:

```
prop_business_logic() ->
  ?FORALL(Commands, commands(?MODULE),
    begin
      true = invariant(),
      {_History,_ State, Result} = run_commands(?MODULE,Commands),
      PostCondition = invariant(),
      clean_up(S),
      PostCondition and Result == ok
    end).
```

The reason to check whether the database invariant is satisfied before the test starts is to be able to use the property also on databases that may already be populated with some data before testing. So for every test sequence, each command is evaluated, and its expected result and actual result are matched as described above. When the whole sequence is successfully performed, the database invariant is checked once more, and if it holds in the final system state, the test case passes and a new test sequence is generated to continue. Also, with QuickCheck automatically taking care of counterexample simplification on failure (*shrinking*), the efficiency and effectiveness of the testing process is highly improved.

6.3.3.4 Insurance policies and business rules

The business rules testing method that has just been described has been used in our real case study ARMISTICE to provide greater confidence on a

particularly critical area of the application: the insurance policies business logic.

Insurance policies, as described in Chapter 4, are one of the key elements in ARMISTICE. These formal documents detail all the terms agreed by the parties with regard to objects of interest, hazards and coverage. The interface functions offered by the policies subsystem include operations to, for instance, create a new insurance policy (`create_policy(PolicyNumber)`), a new renewal for a policy (`create_renewal(PolicyNumber,ValidityPeriod)`) or a new supplement (`create_supplement(RenewalCode, Warranties)`) for a given renewal in certain policy, as well as for updating or deleting them, and also for *publishing*[7]them (i.e., change their status, so they are not longer 'under construction' and become visible to other parts of the system), such as `publish_supplement(SupplementID)`, etc.

[7] The creation of insurance elements is a long task that may take days or even weeks, for an expert user.

There are a lot of business rules that affect data related to insurance policies. The following are two examples to which we have applied our testing method:

Business rule 1:

> "*There can only be one renewal under construction for each given insurance policy, and one supplement under construction for each given renewal*."

Business rule 2:

> "*If a supplement has already been published, the corresponding renewal cannot be under construction*."

whose corresponding database invariant tests are:

Business rule 1:

```
business_rule(Connection,S) ->
   RenConsCount =
    db_interface:process_query(Connection,
      "SELECT COUNT(*) "
      "  FROM renewal "
      " WHERE ren_constr IS TRUE "
      "   AND ren_policy IN (SELECT pol_number "
                           " FROM policy ) "
      " GROUP BY ren_policy "),
   SupConsCount =
    db_interface:process_query(Connection,
      "SELECT COUNT(*) "
      "  FROM supplement "
      " WHERE sup_constr IS TRUE "
      "   AND sup_renewal IN (SELECT ren_code "
                           " FROM renewal) "
      " GROUP BY sup_renewal "),
```

```
   ([] == [ R || R <- RenConsCount, R > 1])
     andalso
   ([] == [ S || S <- SupConsCount, S > 1]).
```

Business rule 2:

```
business_rule(Connection,S) ->
   [] == db_interface:process_query(Connection,
      "SELECT COUNT(*) "
      "  FROM renewal, supplement "
      " WHERE ren_code = sup_renewal "
      "   AND sup_constr IS FALSE "
      "   AND ren_constr IS TRUE ").
```

Even though no actual errors were found by applying our testing method to this case study, the thousands of automatic tests that were successfully executed in a matter of minutes grant us a much greater level of confidence in the correctness of the business logic under test. As a side-effect we now also have a formal specification of (part of) the business rules for this system, which simplifies future extensions and additions to the software.

The method we have presented is based on two main points: the concept of *business rule* and the use of a *state machine*. The final aim is to acquire enough confidence in the business logic implementation of a software system not leading the underlying database to an inconsistent state as a result of regular or faulty operation. By "inconsistent state", we refer to a database state in which one or more of the business rules imposed by the business domain of the system are violated.

We have chosen to express the business rules in a combination of SQL and programming language. This combination is powerful enough to express those rules whereas they are at the same time well understood by the engineers implementing the database application. Of course, one could also use other formalisms, like the Object Constraint Language (OCL, defined as part of UML) to express the constraints in. For our method this would require an extra translation of OCL properties to database queries, but this translation could easily be covered by a library.

The QuickCheck state machine primitives are actually used in a non-standard way (as compared to the use described in the integration testing section), which indicates that a new library module specifically devoted to testing data-intensive systems would be in place. Since the most relevant element in a database-intensive application is the database, we have identified the global state of the system with the state of the database, where the interface functions are the operations that can result in a state change. Abstracting from the database content the minimum amount of information needed to conduct

relevant and meaningful tests, we have presented here the different elements of our testing state machine: a command generator that takes frequencies into account, a function to compute the next state, auxiliary data generators to automatically build proper interface function arguments, and finally a validator function to predict the result of a command invocation beforehand. Comparison between expected and actual result is the way to determine if the system is working as intended, and invariant (business rule) checking after each command sequence (test case) execution is the way to ensure business rule compliance at all times.

7
Functional software development methodology

This chapter compiles and discusses the different aspects of the functional software development methodology that has been the story line of this work, as explained stage by stage so far. Abstracting from the details of our particular study case, relevant contributions, as applicable to other functional application developments, are presented.

7.1 Purpose

The main purpose of this dissertation has been to explore a new software development methodology, including all steps of software construction, in order to propose significant improvements that can lead to a better software engineering experience, both for the developer and the final user.

One way of pursuing the development of better software is by designing and formulating effective methods and techniques that contribute to software quality assurance. Two main factors have the biggest influence over quality: knowledge and management. By increasing our comprehension about business domains and software properties, and by wisely steering the transformation process from requisites into functionalities, we can protect products and activities from different aspects of failure.

The software development community is currently undergoing a hot debate about whether or not software construction is an engineerable activity [148,

149]. Focus has shifted from procedures and documentation to people and skills, and together with that change of concerns has come less interest in formerly worshipped properties such as repeatability of processes, risk constraint, and dependable control. However, software needs to be, more than ever, usable, efficient, maintenable, reliable, secure, safe,... Through the years, the different life cycles that have come and gone were never too specific about how to achieve these properties. While the heterogeneity, delivery, and trust challenges remain open, rigour and formality are losing attention.

All in all, conceiving software development as a mere experimental and artistic expression is no less debatable. Whereas creativity might be a bonus when building a solution for a problem, it is the quality and usefulness of such a solution which is required. Among all the possible ways of addressing the many challenges of software development, only those which are not only clever, but readable, maintenable, scalable, testable, and documented, succeed. We can list a number of common characteristics among software development activities and other engineering disciplines, including problem definition as starting point, creation of models to contrast the engineer's understanding of the problem, feasibility studies to verify the viability of design candidates, importance of design as a central activity, creation of plans for building the product, inspections during creation, verification of requirements fulfilment, and ongoing interplay between theoretical abstract knowledge and practical experience knowledge.

The fact that society has come to accept software failure at certain levels, is indeed a consequence of bad software engineering. In spite of that, the same society does not take software failure as an option in highly critical applications such as medical equipment or aircraft navigation systems.

Consequently, we present here the summary of our experiences as promising recommendations whose main argument continues to explore the direction software development has been travelling for years: the approach of more complex problems using higher-level strategies. To complement the revolution that is giving prominence to professionals at the expense of processes, an innovative development methodology which turns the spotlight on the *what* instead of the *how* along its different stages is proposed.

7.2 Phases

In this section we will refer to the different activities that take place during the development of a given software project. According to each task's inputs, objectives, and outputs, the proposals and methodologies that have been highlighted during the previous chapters will be gathered together and abstracted for general use. The sequential structure in which they will be addressed does not imply any order in their actual fulfilment.

7.2.1 Requirements analysis and system design

The translation of user needs or problem description into specific functionalities and services is one, if not the most, critical step in software development. No matter how efficient an implementation is or that a product presents no bugs: if it fails to meet its expectations or does not satisfy the original necessity, it would be a failure. Even if the deviation is detected soon enough to be amended, corrective actions to deal with bad requisites compilation or interpretation are among the most expensive efforts. For these reasons, not enough stress can be placed at this stage.

Requirements elicitation and system design

The correct interpretation of client demands and wishes, as well as the detection and identification of the relevant details about the business domain properties and the operative processes, should be faced as a joint work between developers and future users. The implication of all parties is an important ingredient for a successful recipe, which turns essential in complex environments that require advanced knowledge and understanding of concepts and activities. Elicitation techniques based on **structured and unstructured interviews**, attended by **multidisciplinary teams** are a good approach that must lay the foundations for a firm relationship and collaboration throughout all the project life.

After the requirements are gathered, and once the main purpose and objectives are clear, system architecture, environment, and main components must be depicted. This stage involves two different activities: conceptual design and technical design. For the first one, **UML standards** represent a powerful, expressive, and flexible way of articulating, communicating, and documenting the abstract structure that will be given to the specific needs as they are modelled and turned from requirements explanations to agents and components properties and behaviour. In particular, our experience in the ARMISTICE case study revealed UML class diagrams as a good communication tool, above the rest of the UML tools, involving a technical level which is easy to reach for non-technical users and clients.

A complete perspective is to be maintained during technical design, paying attention not only to the active elements, their attributes and abilities, but also to the whole collaborative picture. **Design patterns** are excellent resources in this process, as broadly-spread compoundable artifacts which are known to present recognisable solutions to common problems. A wide range of design patterns where used in ARMISTICE's both server and client design, from architectural patterns such as Model-View-Controller or Layers (which enabled us to use specific techniques during testing), to creational patterns such as Singleton objects or objects Pools for specific services in the application backbone, Value Objects and Data Access Objects for persistent storage access, and multiple behavioural and structural patterns in the user interface. On the

other hand, experience as well as specialised knowledge will assist professionals to choose the most suitable technological alternatives.

Efforts in this area, related to the contents of Chapter 4, have been published in different journals and international forums:

Environment-independent methodology for accessing external data sources [150]. Laura M. Castro, Víctor M. Gulías, Carlos Abalde, and Javier París. *WSEAS journal on Transactions on Information Science and Applications*, 2008.
Database access and patterns in Erlang/OTP [151]. Laura M. Castro, Víctor M. Gulías, Carlos Abalde, and Javier París. *Proceedings of the 8th International Conference on Applied Informatics and Communications*, 2008.
A new Risk Management approach deployed over a client/server distributed functional architecture [67]. Víctor M. Gulías, Carlos Abalde, Laura M. Castro, and Carlos Varela. *Proceedings of the 18th International Conference on Systems Engineering*, 2005.

TABLE 7.1. Publications related to system design.

Model review and formalisation

Once the software project has a determined set of goals and there is a prototype design, either just as the system intended internal and external structure in the form of UML diagrams, or including a proof-of-concept, our recommendation is to replace the sensible model review by a more serious task in which we go beyond a model review and formally validate the coherence between engineering plans and clients wishes before proceeding any further.

The suggested strategy for doing so is to address the **formalisation of the core concepts** of the system-to-be as a series of **declarative statements**. This extra effort in re-describing the results of the previous stages in a rigorous mathematic-like style is intended to act as early validation of the elicited abstractions, and hopefully help detect subtle misconceptions that may lead to gaps or faults in the software, improving and fine-tuning its model. Not as expensive as actual prototyping, such statements will not only increase confidence in proper knowledge acquisition, they could also be re-used later if combined with the use of functional programming and automatic validation, providing a valuable tool for traceability. One could argue that, instead of going through an intermediate step as this formalisation, software constraints such as QuickCheck's properties could be written down already at this moment, directly inferred from the analysis documents. However, that would not only require an expert in that tool to perform this task, which may or may not be

feasible, but more importantly, it would not be a neutral language to present to a final user or client. On the other hand, a QuickCheck user should be able to very easily extract QuickCheck properties from this formalised system description.

This proposal has been found interesting by the scientific community, and has taken the shape of the following publications:

Managing the risks of Risk Management [64]. Laura M. Castro, Víctor M. Gulías, Carlos Abalde, and J. Santiago Jorge. *Journal of Decision Systems*, 2008.
Formalisation of a Functional Risk Management System [68]. Víctor M. Gulías, Carlos Abalde, Laura M. Castro, and Carlos Varela. *Proceedings of the 8th International Conference on Enterprise Information Systems*, 2006.

TABLE 7.2. Publications related to model review and formalisation.

7.2.2 Implementation

Historically, time and resources devoted to a software project have been moving from the implementation phase to both requirements analysis and design, and validation and maintenance. Many development frameworks speed up this task by improving the coding experience, and even automatically generating source code from specifications and models in some cases. Still, this is not applicable to a whole system or application, so concerns about how to improve this task remain.

As a natural, but yet to be taken, step in the evolution of software engineering, this dissertation presents the **functional paradigm** as a feasible means of easing and improving the efficiency of the implementation phase.

Declarative programming represents the abstraction from sequences of orders and explicit state management to higher-level algorithm description. The long-time alleged tie between specific analysis and design approaches (such as object-orientation) has been demystified: functional languages have been successfully combined with several paradigms. In fact, this multi-paradigmatic view is closer to the real world, where passive objects and active agents seamlessly collaborate everywhere, performing complex tasks. The key property that the functional paradigm offers is the higher abstraction level at which implementation is carried out, which enables to tackle more complex problems and reduces the distance between specification and modelling, and coding, thus preventing many errors usually introduced by such a gap.

Two relevant publications support the interest and significance of this view:

Erlang/OTP framework for complex management applications development [152]. Carlos Abalde, Víctor M. Gulías, Laura M. Castro, Carlos Varela, and J. Santiago Jorge. *Proceedings of the 3rd International Conference on Web Information Systems and Technologies*, 2007.
ARMISTICE: an experience developing management software with Erlang [66]. David Cabrero, Carlos Abalde, Carlos Varela, and Laura M. Castro. *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, 2003.

TABLE 7.3. Publications related to system implementation.

7.2.3 Verification and validation

The increasing relevance and attention that software quality concerns have been gathering have also given rise to all kinds of verification and validation activities. As the complexity of problems addressed by software grew, so did the software itself and, consequently, testing tools and procedures. The objective is to improve the efficiency and effectiveness of the testing techniques in this new, continually evolving context of sophisticated software systems and applications.

The recommendations given for the previous stages convey some benefits that extend to this one, enabling powerful, more ambitious testing resources and methodologies. In particular, the use of a functional paradigm provides a well suited environment for **model-based testing**, since abstract models are described in a significantly close manner to the way software is actually written. Besides, the absence of side effects and observational equivalence properties present the most convenient context for **automatic testing**. Thanks to this, meaningful confidence can be gained in spite of not applying formal verification methods.

Below we summarise the different testing approaches that have been experimented with and exercised at various validation levels during the development of this thesis, as seen in Chapter 6.

Unit testing of data types using properties

The first contribution of this dissertation in the field of validation is the introduction of a **methodology to test data types**. With this methodology, the automatic testing tool QuickCheck can be used in a more structured way, giving more assurance to tests results. The methodology has been shown to work well for Erlang-based data types, but the language is not a limiting factor, as some recent experiments with C data types demonstrate.

Our validation technique is based on checking whether a data type implementation is equivalent to a trusted model, thus holding equivalent properties. Hence, to apply it, a **model for the data type** must be defined, followed by the creation of **as many model-equivalence checking properties as available operations** in the data type. Important recommendations include working with symbolic values instead of real values, to keep independent of internal representation of the data type. To run the automatic test generator on the defined equivalence properties, data type generators are needed, which must cover all the data structure and include all possible data type constructors (i.e., not only constructors, but all data type operations producing values of the data type as a result). Also, exception cases testing must be placed at the relevant properties, hence keeping the generated values well-defined. Finally, the inclusion of customised shrinking preferences may be in place, to help automatically reach a simplified counterexample.

Even though one may expect data types to be rather simple pieces of software, this methodology proved that failures can be detected even in software considered very stable and used in production for several years. Used on data types from our risk management information system case study, it showed a generally applicable, simple, and fruitful recipe to follow, as acknowledged by the research community:

Testing Erlang data types with Quviq QuickCheck [137]. Thomas Arts, Laura M. Castro, and John Hughes. *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, 2008.

TABLE 7.4. Publications related to verification and validation.

Integration testing using a state machine model

As previously said, during the testing stage several types of checks can be performed on a system or software application. When the software to be checked is composed or structured in different modules or components, they should not only be tested on their own (to ensure they do perform correctly) but also in combination (to ensure they do interact properly). Integration testing, then, ensures that when a service is requested from one component, which relies in other component to perform the operation, the former invokes the right methods from the latter.

In this regard, the second contribution of this dissertation is a **state machine-based integration testing methodology**. Using a state machine for test case generation and execution means generating random sequences of state transitions (representing the component operations to test) from a given initial state, each of them according to certain preconditions, invoking them updating the internal state, and then checking some postconditions. Therefore, to apply this technique, the internal state structure and its contents as initial state

must be defined, as well as the set of operations to test and their preconditions and postconditions, and how the execution of each operation may modify the contents of the internal state. The important observation here is that, since integration testing does not check functionality, but rather only interaction, the second component is not actually needed: a replacement that offers the same interface and mimics the same kind of answers is enough.

This approach is aligned with the principles of black box testing, meaning that random order and argument values on functions invocation is assumed. Consequently, preconditions in the state machine are kept as small and simple as possible (i.e., ideally, empty). As for the postconditions, they are the place to check if a component invokes the proper operations from another. Since the second one needs only be a dummy component, it must register all accesses to its interface functions and provide a way to retrieve that information. Such an invocation trace is recovered to verify that only correct interactions take place in all possible scenarios.

Successful execution of tests produced with this methodology provides confidence on that for each operation in the tested component, the right operations in the external component are called, and in the expected order. Several publications have been derived from this research effort:

A practical methodology for integration testing [153]. Laura M. Castro, Víctor M. Gulías, Carlos Abalde, and Javier París. *Lecture Notes in Computer Science*, 2009.
Testing integration of applications with QuickCheck [154]. Laura M. Castro, Miguel A. Francisco, and Víctor M. Gulías. *Extended Abstracts of 12th International Conference on Computer Aided Systems Theory*, 2008.
Applications integration: a testing experience [155]. Laura M. Castro, Víctor M. Gulías, and Miguel A. Francisco. *Proceedings of 6th Workshop on System Testing and Validation*, 2008.

TABLE 7.5. Publications related to verification and validation (ii).

Data integrity validation via business rules testing using a state machine model

Last but not least, as part of this work, research in the area of business rules testing has been also performed. The previously unaddressed challenge of validating directly the core axioms of an application business logic, which is specially relevant to increase confidence on data integrity, has been explored, and a **method** suitable for **testing data-intensive applications** is proposed.

Business rules are domain constraints that are commonly assumed to be satisfied by a system or application, but due to their abstract and high-level nature, they are hardly ever located at one single point in the implementation. Besides, the interaction of different services and functionalities may inadvertedly break the data consistency demanded by such business rules. Using QuickCheck again, we develop a testing process that can be, nevertheless, easily exported and reproduced in other environments. The procedure consists of a combination of design discipline and test specification techniques which, assuming that the application has been architectured with a layered structure and that all access to data from the business logic layer is conducted via a specific persistence layer, suggests formulating the business rules as invariants in the form of data-inspecting queries. A state machine for testing will then have the exported functions from the public system interfaces (access points to the business logic) as transitions, a minimum internal state to be able to generate operations which refer to the same entities (i.e., object keys or similar), and plain true preconditions and postconditions, to allow the simulation of any client's interaction and perform black and white testing at the same time. To diagnose a testing case as successful or not, the actual state of the data source is consulted to determine the success or failure of a given call beforehand, and is then contrasted with the real execution of the function, validating whether the result of the interface service invocation corresponds to what was expected. Last but not least, invariants (i.e., business rules) are checked after each testing sequence, to show whether data consistency is still maintained.

Contributions in this area are yet to be published.

Testing data consistency of data-intensive applications using QuickCheck
Laura M. Castro, and Thomas Arts.
To be published, 2010.

TABLE 7.6. Publications related to verification and validation (iii).

7.3 SWOT analysis

We conclude this chapter by performing a SWOT analysis on the functional software development methodology that is proposed here. SWOT analysis is a well-known management method which tries to evaluate the *strengths*, *weaknesses*, *opportunities*, and *threats* involved in a project or business prospect [156]:

Strengths are internal factors that benefit a given objective. In our context, they are the properties of our development methodology that most influence the construction of quality software.

FIGURE 7.1. SWOT Analysis

Weaknesses are also internal factors, but rather than assisting the objective they represent a menace. In our development methodology, this means the properties that can turn against us and be an obstacle in our way to good software.

Opportunities play a similar role to that of strengths in SWOT analysis, but they are external aspects instead of internal. In other words, they are the circumstances that surround software development nowadays, which can support the success of our methodology.

Threats are also external elements or situations that stand in the way of a project goal. For our proposal, these are the circumstances that could impede its successful application.

Essential for strategic planning, the identification of the internal and external factors that may favour or jeopardise an activity or plan is intended to get the most out of the first while keeping the latter under control during the pursue of an objective.

Undoubtedly, the strongest feature of this methodology lays around the use of the functional paradigm and all the benefits that it brings with it. This ranges from reduction semantics that avoid side effects such as aliasing (coherence problems due to multi-referencing) and thus favour the untroubled introduction of parallelism, for instance, to the expressive power of high-order functions or

the cost-effectiveness of lazy evaluation. Thanks to these attributes, reduced effort is necessary to transform program descriptions into source code. Also, the nature of functional programmed code is not only prone to optimisations, but also an excellent context for the introduction of automated tools for behaviour prediction, code enhancement, and verification. Hence, advantages range from the implementation to the maintenance stages.

However, all programming paradigms and languages can be badly used as to become counter-productive. In the case of functional programming, there are some tasks that may be poorly suited for it, such as low-level driver and controller coding, or whenever fine-grain control over parallel hardware is required. Since hardware nature is essentially imperative this cannot really be avoided; it constitutes a serious weakness, at least for the time being.

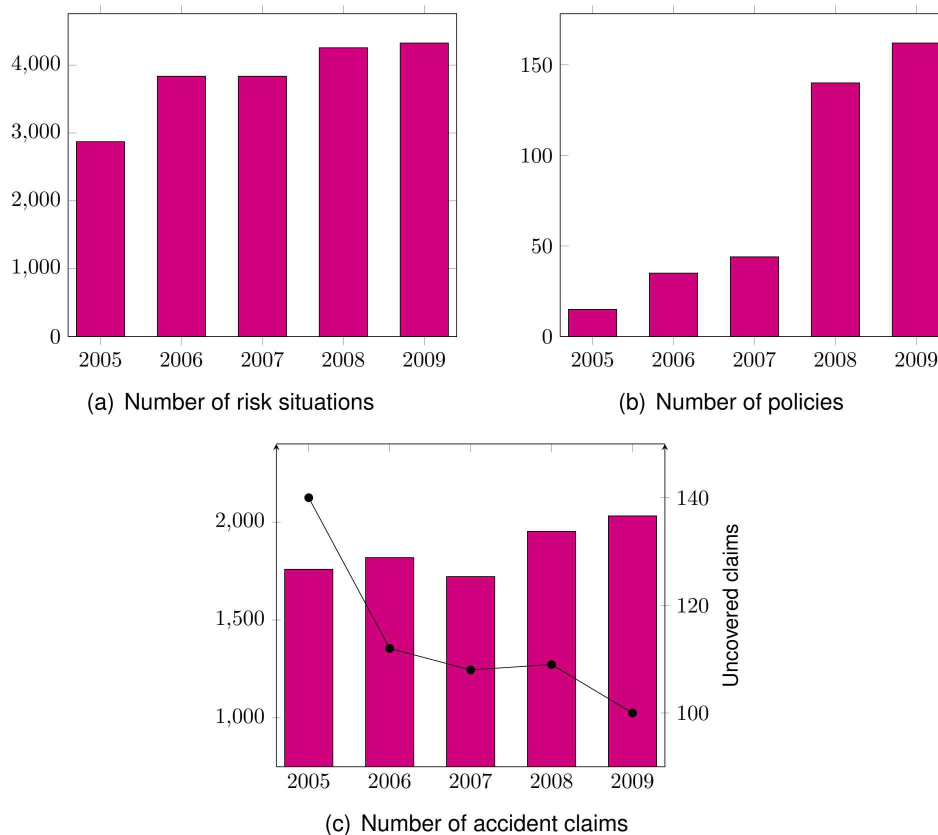
Still, the fields and challenges that software faces grow broader day by day, as do the kinds of services and functionalities that are required from applications and systems. Our society has not only embraced technology, and in particular software, it has reached a point in which it requires it at all levels and scales. In this context, the ability to efficiently and effectively face newer and more complex domains is a great opportunity, and has a specific weight that cannot be overlooked.

Last but not least, the proposal of new ways of doing things, approaching problems, and performing common tasks, always needs to break down the barrier of the established *status quo*. In particular, changes in the way we think and act face the force of habit and routine. Even harder than that, wrong ideas and misconceptions need to be removed, such as the belief that functional programming is only well suited for non-practical issues, academical purposes, and research projects.

7.4 Case study evaluation

For a few years now, our case study application, ARMISTICE, has been successfully deployed in the risk management department of a large holding enterprise, where it has replaced the rudimentary solutions (specifically, a number of extremely complicated spreadsheets) that were used before for the daily job of the staff. The suitability and adaptability of the development methodology that was followed and the implementation technologies that were used to build the system have been demonstrated, as it has been possible to easily model all the insurable elements, policies, and hazards, and provide a full set of complex functionalities involving the lifespan of insurable objects, the complete modelling of policies, and the detailed management of accident claims. The amount of information and the degree of control that users are provided by the system is a differential factor regarding other RMIS systems, and it has made it possible to detect mistakes in the original composition of some real policy clauses.

Diagrams below show some statistics concerning the volume of information that ARMISTICE has been managing during its years in production, as well as its evolution.



(a) Number of risk situations

(b) Number of policies

(c) Number of accident claims

FIGURE 7.2. Real evolution of business objects in ARMISTICE

As can be seen in Figure 7.2, (a) and (b), both the number of risk situations and contracted policies have grown from year to year, in the first case, at a quite stable pace, which very likely reflects the natural expansion of the client company. Regarding contracted policies, however, there is a great increase in the number of policies managed by the system each year with respect to the previous one from the moment the software was deployed. We interpret this as a reflection of the client's level of confidence in the risk management product, as the contracted policies are consistently translated into their ARMISTICE counterpart, and paperwork is steadily substituted by ARMISTICE's processes. Last but not least, a third diagram (Figure 7.2 (c)) shows the evolution of processed claims. We can appreciate that the overall number of claims that remain uncovered, that is to say, whose losses cannot be recovered by referring to any of the existing insurance policies, has been decreasing since ARMISTICE's introduction, in spite of the increasing number of risk objects and claims. We would definitely need more specific data corresponding to fur-

ther years in order to know if this is a consistent trend. If so, this could mean that the department's risk management policy, now improved and assisted by ARMISTICE, is helping the company to reduce the number of accidents that translate into non-recovered losses. Though these measurements cannot be claimed as 100% accurate metrics, they constitute informative indicators that can be regarded as very positive feedback about the success of the system.

8 Conclusions

In this final chapter, we go over the contributions that have been made, explain the lessons that have been learned, and discuss some future work to be pursued.

8.1 Contributions

This PhD. thesis summarises an applied research effort in which a real case study has served as basis to conduct software engineering research. Our work led to the extraction of some interesting points about an alternative way of developing software, as well as to the exploration and elaboration of several specific methods and procedures, that can now confidently be advised to be applied in other developments as well.

The functional paradigm has proved, this time with a complex management domain as environment (i.e., risk management), that when regular software production enters especially complex grounds, the possibilities that declarative programming provides are of great help. Hence, the methodology that has been followed and explained here is a step forward into formalising and establishing such a development environment as a serious alternative to more traditional developments, that cannot face the same sort of problems to the same extent and with comparable effort.

The correctness of the system under development being one of our main concerns, the most relevant and practical contributions of this work lay in the testing field, where three different levels have been approached, and three com-

plementary strategies have been proposed. While being all rooted in model-based testing, each of these strategies addresses a different testing subject and distinct testing aims are exercised (namely, verification: unit testing, integration testing; and validation: business rules testing). Also, in spite of relying on the use of a specific automatic testing tool (i.e., QuickCheck), they have been abstracted and formulated in order to be generally applicable.

Using test case generation tools is not a trivial task, though. A too naïve approach can convince ourselves we have tested enough when we are actually missing a lot or are even not really proving anything relevant. In the development of the proposed verification and validation strategies, a serious effort has been made to avoid these pitfalls. The followed processes have been exhaustively explained here, and each step has been justified in the light of the possible problems to face (wrong approaches, obscurity of dealing with unusual scenarios, failure to achieve a complete test coverage, unsatisfactory fault report). As a solution to overcome them, model definition, use of symbolic values, careful data generators definition, or self-defined counterexample shrinking rules, for instance, have been explored in depth and successfully tried out on different parts and aspects of our case study application, in some cases even to find errors whose symptoms had occurred for some time but whose scarce bug reports were either not well understood or not sufficiently informative to diagnose them.

The proposed testing methodologies make use of the definition of properties and state machine behaviour description as central elements, which bring important advantages. First of all, the use of such abstractions enables the automatic generation of all kinds of testing sequences, only limited by the properties or state machine definitions themselves. These testing sequences simply adjust to the set of constraints specified as model properties or state machine specifications, hence including not only the kinds of operation sequences that will usually be tested, but all kinds of sequences, no matter how strange or improbable they might seem. Having the possibility of testing all sorts of unlikely cases is very useful to find the sorts of errors that are more expensive to fix (both in terms of time and effort), most of all if found after releasing or deploying the system into a production environment.

The criterion to determine if a test case has been successful or not varies accordingly with the testing scope. For unit-testing, an equivalent model is defined and used as reference for expected result of test case execution. For integration testing, external components are replaced by API-equivalent dummy ones, and communication (invocation) traces are inspected. Finally, business rule consistency testing relies on data-related domain invariants specified as SQL-statements. These activities improve both system confidence and knowledge at the same time, since they require careful identification and description of properties and behaviour at various levels.

To conclude, these pragmatic contributions have been complemented with recommendations about knowledge elicitation and knowledge reassurance processes, where the formalisation of business concepts links perfectly the requirements gathering, design, and implementation stages, representing a valuable tool for traceability, and even filling in the gap between the real world concepts and the source code, provided that functional programming is in place, and consequently improving the development experience and the developed product.

8.2 Lessons learnt

Improvements in software development are a never-ending quest. Resources increase, performance enhances, tools appear, technologies advance. As a result, not only the products we create, but the building process itself, must be modified to take the most advantage of the existing capabilities at the current moment. After all, the quality of the result is very often heavily influenced by the quality of the production procedure.

For years, functional programming in particular, and the declarative paradigm in general, did not have a significant role in industry. However, this is clearly changing, as reflected, for instance, in the interest, presence, and relevance that commercial users and developers are gaining in traditionally theoretic and academic-oriented scenarios such as the International Conference on Functional Programming (ICFP, possibly the most important international conference on the subject) and its satellite events. Telecommunication companies, with Ericsson leading, are using Erlang for critical control software; Internet applications and web services providers are also taking advantage of this and other functional languages; numerous small and medium size companies demand more and more professionals with experience or background in declarative programming, for a variety of projects which range from massive distributed systems (i.e., Facebook chat), to data acquisition and real-time monitoring systems (i.e., Finish Meteorological Institute), including electronic payment systems or market analysis tools in the financial sector (i.e., Credite Suisse, Deutsche Bank) [157, 158]. This trend has been reflected by the most relevant programming online indexes as well [159].

At the same time, the recognition of the importance of verification and validation tasks within the software life cycle, is also growing. Still, its unarguable complexity and the great amount of time and resources usually needed to perform formal verification or even plain testing properly, sometimes together with (or due to) the lack or unawareness of powerful and versatile tools, implies that, in practise, these activities are often underestimated and diminished, or just simply ignored and skipped, sometimes due to the client's demands or hard time-to-market constraints.

The background of this work has been our case study ARMISTICE, a very convenient risk management process support tool, with decision making assistance abilities, not bounded to a specific business area thanks to its abstraction capabilities, and meta-information definition and use. This case study is a successful experience report that shows that, irregardless of the complexity of a given domain, existing engineering techniques, methodologies, and procedures are powerful enough to overcome initial concerns. A key step in the way of applying new technologies in such scenarios is domain knowledge elicitation from the experts. Here we have shown how the available standard notational solutions and well-known development life cycles perfectly apply and favour good results.

The use of a declarative approach in order to build general-purpose software has helped to reduce the gap between the analysis and design stages, and the implementation stage. Problems and concepts are commonly abstracted and translated into definitions and functionality descriptions, using high-level languages and tools such as UML. Important software errors or *bugs* may be introduced when diagrams and models are translated again into specific software components and source lines. However, using a higher-level implementation language decreases the virtual distance between design and implementation, and hence diminishes the chances of errors appearing due to this necessary step. As a side effect, more complex challenges can be addressed, and more complex systems can be designed and built with reasonable effort.

Additionally, the testing procedures that have been shown here have also important advantages. First of all, the use of the automatic test-case generator QuickCheck allows us to perform many random sequences of operations just after writing down a few properties. The way the properties or behaviour under test are specified allows QuickCheck to generate not only the kind of operation sequences that will usually be tested, but all sequences (no matter how strange or improbable they might be) that adjust to the set of indicated constraints. Having the possibility of finding such errors before an application or component is released or sent to a production environment is extremely valuable and might potentially save a lot of time and effort. Also, QuickCheck not only automatically generates a great amount of random test cases, but also provides useful information when a fault is detected. Whenever QuickCheck finds an error, it returns the exact trace of operations that led to that error; even more, it also generates the smallest equivalent trace which produces the same error. These traces are generally of good help in identifying and locating the problem in the source code, and extremely helpful if this situation is compared with having just a brief description of a strange behaviour given by a user once the system or application is working in a real environment.

Different testing scenarios have been addressed, among the most common in any software application. Most modern systems are structured in different separate modules that work together to offer full services or provide complex

functionalities. Most management systems embody a whole set of working rules (business constraints) that are hardly ever explicitly placed at a single point in the system. Most complex systems need to define their own data types, to have an efficient way of dealing with their specific data structures. In these situations, integration, business rules, and data types testing are essential aspects to check. Hence, three different testing strategies have been developed, in order to provide a generally-applicable, systematic way of approaching these particular testing tasks, which we consider of great interest.

To sum up, this thesis is a claim about the declarative paradigm as a greatly useful tool that improves the construction of complex software, and an exposition of several techniques which, in such context of functional programming, allow to address development, verification, and validation at different levels, with reduced effort and contrasted results.

8.3 Prospects: open research lines

As much progress as we are making in introducing new technologies in almost every aspect of our lives and jobs, there seems to be a disconcerting lack of generic, flexible, powerful tools in especially complex domains, as it is the case of Risk Management. This is clearly not due to an absence of need for them, since complex tasks can enormously benefit from comprehensible tools to improve performance and user experience, and to increase the level of knowledge, control, and overall results.

Concerning our case study, there is still further work to be done. A few ideas on this include additional customisable reports, for example, perhaps in the same way the system already deals with risk situations and hazards definition: through the use of meta-information. Another very interesting line of inquiry would be that of architectural and functional pattern detection for these sort of highly critical applications. The effort carried out to meticulously analyse the domain and extract the relevant information that was then written down as a model design, led to the gathering of the kind of expert knowledge that would be needed for such a task. Locating behaviour or structural similarities between these kinds of knowledge-intensive software systems could be really interesting and open a whole set of research possibilities.

The deployment process has revealed some framework improvements like the definition of dynamic properties in the groups of objects of interest, that could be done by introducing the concept of formula (λ-expressions) inside the base data types ($Type$ set), as is already being done in other parts of the system. In this way, even more flexible behaviours could be performed. Another prospect involves the definition of a standardised language to encode all of the behaviour of a policy and not only its formulae and restrictions.

There is also room for improvement at the reliability-ensuring activities: software verification and validation techniques and tools to ensure the compliance

and persistence of the intended properties and behaviours of applications are also a very important topic to keep looking further into. Automatic test case generation software such as QuickCheck is a very powerful tool to use when aiming for better software, which also means software submitted to more complete validating procedures. QuickCheck improves the testing process not only by making it easier and faster, but also more exhaustive. The testing strategies proposed here can still be enhanced. For instance, the methodology for testing data-types could be extended to non-Erlang data types, taking advantage of already existing intercommunication alternatives between Erlang and other technologies such as C or Java. This could be most interesting for multi-language and multi-paradigm inter-operable applications and environments, rather popular in heterogeneous systems.

With regard to the methodology developed for integration testing, the way of contrasting the generated traces and the expected ones could also be improved. Currently, this task is implemented manually considering all feasible situations, which makes the comparison very complex due to all the possible scenarios that must be considered depending on the values returned by the traced functions. Designing a framework for specifying a possible trace and comparing two traces with several possibilities and branches of operations would facilitate the entire process.

Last but not least, the business rule validation methodology could be performed using other formalisms to express the data-related invariants in, like the Object Constraint Language (OCL, defined as part of UML), instead of SQL. In cases where data persistent storage is provided by a relational database, this would require an extra translation from OCL properties to database queries; but whenever alternative storages are used, it would constitute a better, neutral, and standard choice. The mentioned OCL-to-persistent-storage translation (i.e., OCL-to-SQL, for instance) could easily be covered by a specific-purpose library.

Index

business rules, 129, 142, 154

Capability Maturity Model, 19
  Integration, 19

decision support system, 71
design, 35, 149
  formalisation, 44, 150
  patterns, 36

Erlang, 59

ISO
  9000, 9001, 19

meta-information, 30, 39

object of interest, 26
object-oriented
  development, 54

programming
  functional, 4, 21, 151
    patterns, 59
  object-oriented, 22
  paradigm, 4, 21
    declarative, 4, 21, 58
    imperative, 4, 21
    parallel, 5

QuickCheck, 88

requirement, 34
  analysis, 33
  elicitation, *see* ∼ extraction
  extraction, 34, 149
risk management, 25–26

software
  development, 3
    methodology, 11
    process, 11
  engineering, 3, 9
  life cycle, 11
    V model, 13
  testing, 23, 76, 152
    levels, 78
  validation, 23, 77, 129, 152, 154
  verification, 23, 77, 152

testing
  acceptance, 80
  black-box, 83
  dynamic, 80
  installation, 79
  integration, 79, 114, 153
  negative, 87, 126
  positive, 87
  property-based, 90, 152
  random, 90
  state machine, 92, 116, 134, 153, 154
  static, 80
  system, 79
  unit, 79, 152
  white-box, 82

Unified Modeling Language (UML), 35

Bibliography

[1] I. Sommerville, *Software Engineering*, Addison-Wesley, 8th edition, 2006.

[2] F. L. Bauer, Software Engineering, in *First NATO Software Engineering Conference*, 1968.

[3] E. Dijkstra, The Humble Programmer, Communications of the ACM (1972).

[4] P. L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, PhD thesis, University of California at Berkeley, 1990.

[5] C. Reade, *Elements of Functional Programming*, Addison-Wesley, 1989.

[6] T. Neward, The Success of Java, O'Reilly's On Java (2002), `http://www.oreillynet.com/onjava/blog/2002/09/the_success_of_java.html`.

[7] J. E. Tomayko and O. Hazzan, *Human Aspects of Software Engineering*, Charles River Media, Inc., 2004.

[8] R. L. Glass, *Software Engineering: Facts and Fallacies*, Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] H. Erdogmus, Essentials of Software Process, IEEE Software **25**(4), 4–7 (2008).

[10] P. Bourque, R. Dupuis, A. Abran, J. W. Moore, L. Tripp, and S. Wolffe, Fundamental principles of software engineering - A journey, Journal of Systems and Software **62**(1), 59–70 (2002).

[11] I. Sommerville and G. Kotonya, *Requirements Engineering: Processes and Techniques*, John Wiley & Sons, Inc., 1998.

[12] D. Hatley, P. Hruschka, and I. A. Pirbhai, *Process for system architecture and requirements engineering*, Dorset House Publishing Co., Inc., 2000.

[13] B. S. Blanchard and W. J. Fabrycky, *Systems Engineering and Analysis*, John Wiley & Sons, Inc., 3rd edition, 2003.

[14] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*, Artech House, Inc., 2002.

[15] R. Black, *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*, John Wiley & Sons, Inc., 2002.

[16] T. M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, John Wiley & Sons, Inc., 1996.

[17] J. Rost, Software Engineering Theory in Practice, IEEE Software **22**(2), 96–95 (2005).

[18] R. J. Botting, Theory and practice of software engineering, in *Proceedings of 17th ACM Annual Conference on Computer Science*, pages 481–481, ACM Press, 1989.

[19] W. W. Royce, Managing the Development of Large Software Systems, in *Proceedings of IEEE WESCON*, pages 1–9, 1970.

[20] S. L. Pfleeger and J. M. Atlee, *Software Engineering – Theory and Practise*, Prentice Hall, 3rd edition, 2003.

[21] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975.

[22] C. Larman and V. R. Basili, Iterative and Incremental Development: A Brief History, IEEE Computer Archive **36**(6), 47–56 (2003).

[23] B. Boehm, A Spiral Model of Software Development and Enhancement, ACM SIGSOFT Software Engineering Notes **11**(4), 14–24 (1986).

[24] J. Martin, *Rapid Application Development*, MacMillan Publishing Co., Inc., 1991.

[25] D. Cohen, M. Lindvall, and P. Costa, An introduction to agile methods, Advances in Computers **62**, 1–66 (2004).

[26] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 6th edition, 2007.

[27] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[28] R. H. Thayer, *Project Manager's Guide to Software Engineering's Best Practices*, IEEE Computer Society Press, 2002.

[29] J. Shore and S. Warden, *The Art of Agile Development*, O'Reilly, 2007.

[30] J. Hunt, *Agile Software Construction*, Springer-Verlag New York, Inc., 2005.

[31] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 2nd edition, 2006.

[32] A. Fuggetta, A Classification of CASE Technology, IEEE Computer Archive **26**(12), 25–38 (1993).

[33] I. Vessey and A. P. Sravanapudi, CASE tools as collaborative support technologies, Communications of the ACM **38**(1), 83–95 (1995).

[34] H. Watts, *Managing the Software Process*, Addison-Wesley, 1989.

[35] J. R. Persse, *Implementing the Capability Maturity Model*, John Wiley & Sons, Inc., 2001.

[36] Standard CMMI Appraisal Method for Process Improvement (SCAMPI^{SM}), http://www.sei.cmu.edu/library/abstracts/reports/06hb002.cfm, Aug. 2006.

[37] D. M. Ahern, R. Turner, and A. Clouse, *CMMI Distilled: A Practical Introduction to Integrated Process Improvement*, Addison-Wesley Longman Publishing Co., Inc., 2003.

[38] M. B. Chrissis, M. Konrad, and S. Shrum, *CMMI: Guidelines for Process Integration and Product Improvement*, Addison-Wesley Professional, 2nd edition, 2006.

[39] R. Bamford and W. Deibler, *ISO 9001: 2000 for Software and Systems Providers: An Engineering Approach*, CRC-Press, 2003.

[40] R. T. Futrell, L. I. Shafer, and D. F. Shafer, *Quality Software Project Management*, Prentice Hall, 2001.

[41] J. S. Osmundson, J. B. Michael, M. J. Machniak, and M. A. Grossman, Quality management metrics for software development, Information and Management **40**(8), 799–812 (2003).

[42] R. W. Floyd, The paradigms of programming, Communications of the ACM **22**(8), 455–460 (1979).

[43] L. G. Samaraweera and R. Harrison, Evaluation of the functional and object-oriented programming paradigms: a replicated experiment, ACM SIGSOFT Software Engineering Notes **23**(4), 38–43 (1998).

[44] S. R. Schach, *Object-Oriented and Classical Software Engineering*, McGraw-Hill, 2007.

[45] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley Longman Publishing Co., Inc., 2004.

[46] E. Stiller and C. Le Blanc, *Project-Based Software Engineering: An Object-Oriented Approach*, Addison-Wesley Longman Publishing Co., Inc., 2001.

[47] M. Shannon, G. Miller, R. J. Prewitt, and S. Loveland, *Software Testing Techniques: Finding the Defects that Matter*, Charles River Media, 2004.

[48] W. E. Lewis, *Software Testing and Continuous Quality Improvement*, Auerbach Publications, 3rd edition, 2008.

[49] G. D. Everett and R. Jr. McLeod, *Software Testing: Testing Across the Entire Software Development Life Cycle*, Wiley-IEEE Computer Society Press, 2007.

[50] J. Kasurinen, O. Taipale, and K. Smolander, Analysis of Problems in Testing Practices, in *Proceedings of 16th Asia-Pacific Software Engineering Conference*, pages 309–315, IEEE Computer Society Press, 2009.

[51] W. Schulte, Challenge problems in software testing, in *Proceedings of 3rd International Workshop on Software Quality Assurance*, page 1, ACM Press, 2006.

[52] D. V. Smoline, Some problems of computer-aided testing and "interview-like tests", Computers and Education **51**(2), 743–756 (2008).

[53] Open Problems in Testability Transformation, in *Proceedings of 2nd IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 196–209, IEEE Computer Society Press, 2008.

[54] E. Kauf, *La Maîtrise des Risques*, Securitas, 1978.

[55] U. Nordblad, Risk Management, in *Risk Management Conference (ICEA)*, 1982.

[56] F. J. Navas Oloriz and G. Fernández Isla, Programa de Gerencia de Riesgos en la Empresa, Gerencia de Riesgos **IV**(2) (1986).

[57] The Enterprise Risk Management Annual Conference, http://www.conference-board.org/erm.htm, Aug. 2007.

[58] Middle East Risk Management Annual Congress, http://www.iirme.com/risk/, Aug. 2007.

[59] Risk Minds Annual Conference, http://www.icbi-uk.com/riskminds/, Aug. 2007.

[60] Risk Management Annual Conference, http://www.cboe.com/rmc/, Aug. 2007.

[61] The Institute of Risk Management, http://www.theirm.org/, Aug. 2007.

[62] Risk and Insurance Management Society, Inc., http://www.rims.org/, Aug. 2007.

[63] The Risk Management Association, http://www.rmahq.org/RMA/, Aug. 2007.

[64] L. M. Castro, V. M. Gulías, C. Abalde, and J. S. Jorge, Managing the Risks of Risk Management, Journal of Decision Systems **17**(4), 501–521 (2008).

[65] Coopers and Lybrand, *Los nuevos conceptos del Control Interno*, Díaz de Santos, 1997.

[66] D. Cabrero, C. Abalde, C. Varela, and L. M. Castro, ARMISTICE: An Experience Developing Management Software with Erlang, in *Proceedings of 2nd ACM SIGPLAN Workshop on Erlang*, pages 23–28, ACM Press, Aug. 2003.

[67] V. M. Gulías, C. Abalde, L. M. Castro, and C. Varela, A New Risk Management Approach Deployed over a Client/Server Distributed Functional Architecture, in *Proceedings of 18th International Conference on Systems Engineering*, pages 370–375, IEEE Computer Society Press, 2005.

[68] V. M. Gulías, C. Abalde, L. M. Castro, and C. Varela, Formalisation of a Functional Risk Management System, in *Proceedings of 8th International Conference on Enterprise Information Systems*, pages 516–519, INSTICC Press, 2006.

[69] S. G. Schreiber, *Knowledge Engineering and Management*, MIT Press, 2000.

[70] S. Kendal and M. Creen, *An Introduction to Knowledge Engineering*, Springer-Verlag New York, Inc., 2006.

[71] K. H. Bennett and V. T. Rajlich, Software maintenance and evolution: a roadmap, in *Proceedings of Conference on The Future of Software Engineering*, pages 73–87, ACM Press, 2000.

[72] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modelling Language Reference Manual*, Prentice Hall, 2nd edition, 2005.

[73] E. Braude, *Software Engineering. An Object-Oriented Perspective*, John Wiley & Sons, Inc., 2001.

[74] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Professional Computing Series, Addison-Wesley, 1999.

[75] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1998.

[76] C. Larman, *Applying UML and Patterns*, Prentice Hall, 1998.

[77] W. C. Lim, Effects of Reuse on Quality, Productivity, and Economics, IEEE Software **11**(5), 23–30 (1994).

[78] E. H. Erikson, *Business Modeling with UML (Business patterns at work)*, John Wiley & Sons, Inc., 2001.

[79] B. Boehm and V. R. Basili, Software Defect Reduction Top 10 List, IEEE Computer Archive **34**(1), 135–137 (2001).

[80] M. Fowler, *Patterns of Enterprise Application Architecture*, Signature, Addison-Wesley Professional, 2002.

[81] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.

[82] K. Williams, Using object oriented analysis and design in a non-object oriented environment experience report, in *Proceedings of International Conference on Software Maintenance*, page 109, IEEE Computer Society Press, 1995.

[83] GNOME, GNOME 2.0: An Innovative Platform for Building Advanced Applications, Technical report, Sun Microsystems, 2003.

[84] I. Herraiz, G. Robles, J. J. Amor, T. Romera, and J. M. González, The processes of joining in global distributed software projects, in *Proceedings of 1st International Workshop on Global Software Development for the Practitioner*, pages 27–33, ACM Press, 2006.

[85] B. Shibuya and T. Tamai, Understanding the process of participating in open source communities, in *Proceedings of 2nd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 1–6, IEEE Computer Society Press, 2009.

[86] T. Kühne, *A Functional Pattern System for Object-Oriented Design*, PhD thesis, Darmstadt University of Technology, 1999.

[87] R. Pickering, *Foundations of F#*, Apress, 2007.

[88] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide*, Artima Incorporation, 2008.

[89] R. Lämmel and J. Visser, Design patterns for functional strategic programming, in *Proceedings of 3rd ACM SIGPLAN Workshop on Rule-based Programming*, pages 1–14, ACM Press, Oct. 2002.

[90] V. M. Gulías, A. Valderruten, and C. Abalde, Building functional patterns for implementing distributed applications, in *Proceedings of IFIP/ACM Latin America conference on Towards a Latin American agenda for network research*, pages 89–98, ACM Press, 2003.

[91] LambdaStream, Video on Demand Kernel Architecture (VoDKA), http://www.lambdastream.com/index.php?page=vodka&hl=en_UK, Jan. 2010.

[92] Igalia, SERVAL: Internet software VLAN switch developed in Erlang, http://serval.igalia.com, Jan. 2010.

[93] YAWS: high performance webserver, http://yaws.hyber.org/, Jan. 2010.

[94] Tsung: multi-protocol distributed load testing tool, http://tsung.erlang-projects.org/, Jan. 2010.

[95] Ejabberd: jabber/XMPP instant messaging server, http://www.ejabberd.im/, Jan. 2010.

[96] *Erlang/OTP Documentation*, Jan. 2010, http://www.erlang.org/doc.html.

[97] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in Erlang*, Prentice Hall, 2nd edition, 1996.

[98] F. Cesarini and S. Thompson, *Erlang Programming*, O'Reilly, 2009.

[99] WOOPER: Wrapper for Object-Oriented Programming in Erlang, http://ceylan.sourceforge.net/main/documentation/wooper/, 2007.

[100] eXAT: eXperimental Agent Tool, http://www.diit.unict.it/users/csanto/exat/, 2008.

[101] A. Di Stefano and C. Santoro, eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang, in *Proceedings of AI*IA/Taboo joint Workshop on Objects and Agents*, Pitagora Editrice Bologna, 2003.

[102] A. Di Stefano and C. Santoro, Designing Collaborative Agents with eXAT, in *Proceedings of 13th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 15–20, IEEE Computer Society Press, 2004.

[103] G. Fehér and A. G. Békés, ECT: an object-oriented extension to Erlang, in *Proceedings of 8th ACM SIGPLAN Workshop on Erlang*, pages 51–62, ACM Press, 2009.

[104] R. Carlsson, Inheritance in Erlang, Technical report, Erlang/OTP User Conference, Nov. 2007, http://www.erlang.se/euc/07/papers/1700Carlsson.pdf.

[105] L. M. Castro, Diseño e Implementación de Aplicaciones utilizando Programación Funcional Distribuida: un caso de estudio, Master's thesis, Universidade da Coruña, 2003, http://www.madsgroup.org/staff/laura/academico.html#proyecto.

[106] F. Marinescu, *EJB Design Patterns. Advanced Patterns, Processes and Idioms*, John Wiley & Sons, Inc., 2002.

[107] Open Source Erlang, Aug. 2003, http://www.erlang.org.

[108] Sun Microsystems Java Technology, http://java.sun.com, 2003.

[109] XML-RPC Specification, http://www.xmlrpc.com/spec, 2007.

[110] R. Orfaly, J. Edwards, and D. Harkey, *Essential Client/Server Survival Guide*, John Wiley & Sons, Inc., 3rd edition, 1999.

[111] Software Engineering Institute, Summary Report of Appraisal Results, Technical report, Carnegie Mellon, 2010, http://sas.sei.cmu.edu/pars/pars.aspx.

[112] L. van der Aalst and J. Vink, Testing expensive? Not testing is more expensive!, in *Proceedings of Test Excellence through Speed and Technology International Conference*, pages 1–12, 2008.

[113] G. J. Myers and C. Sandler, *The Art of Software Testing*, John Wiley & Sons, Inc., 2nd edition, 2004.

[114] D. Gelperin and B. Hetzel, The Growth of Software Testing, Communications of the ACM **31**(6), 687–695 (1988).

[115] J. G. Brookshear, *Theory of computation: formal languages, automata, and complexity*, Benjamin-Cummings Publishing Co., Inc., 1989.

[116] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, Cambridge, UK, 2008.

[117] C. Kaner, J. L. Falk, and H. Q. Nguyen, *Testing Computer Software*, John Wiley & Sons, Inc., 2nd edition, 1999.

[118] B. Beizer, *Software testing techniques*, Van Nostrand Reinhold Co., 2nd edition, 1990.

[119] S. Vegas, *A Characterisation Schema for Software Testing Techniques*, PhD thesis, Universidad Politécnica de Madrid, 2002.

[120] R. G. Hamlet, Special Section on Software Testing, Communications of the ACM **31**(6), 662–667 (1988).

[121] L. Damm, L. Lundberg, and C. Wohlin, Faults-slip-through - a concept for measuring the efficiency of the test process, Software Process: Improvement and Practice **11**(1), 47–59 (2006).

[122] K. Claessen and J. Hughes, QuickCheck: a lightweight tool for random testing of Haskell programs., in *Proceedings of 5th International Conference on Functional Programming*, pages 268–279, 2000.

[123] D. Hamlet, When only random testing will do, in *Proceedings of 1st International Workshop on Random Testing*, pages 1–9, ACM Press, 2006.

[124] J. W. Duran and S. C. Ntafos, An Evaluation of Random Testing, IEEE Transactions on Software Engineering **10**(4), 438–444 (1984).

[125] C. Soldani, QuickCheck++, `http://software.legiasoft.com/quickcheck/`, 2010.

[126] T. Jung, Java implementation of QuickChek, `http://quickcheck.dev.java.net/`, 2010.

[127] C. League, QCheck/SML, `http://contrapunctus.net/league/haques/qcheck/`, 2010.

[128] T. Arts, J. Hughes, J. Johansson, and U. Wiger, Testing Telecoms Software with Quviq QuickCheck, in *Proceedings of 5th ACM SIGPLAN Workshop on Erlang*, ACM Press, 2006.

[129] QuviQ A. B., `http://www.quviq.com`, 2008.

[130] J. Hughes, QuickCheck Testing for Fun and Profit, Lecture Notes in Computer Science **4354**, 1–32 (2007).

[131] J. C. King, Symbolic execution and program testing, Communications of the ACM **19**(7), 385–394 (1976).

[132] R. W. Floyd, Assigning meaning to programs, in *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32, American Mathematical Society, 1967.

[133] C. A. R. Hoare, Proof of Correctness of Data Representations, Acta Informatica **1**, 271–281 (1972).

[134] I. of Electrical and E. Engineers, IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985 (Aug. 1985).

[135] J. Barklund and R. Virding, Erlang 4.7.3 Reference Manual, Technical report, Ericsson Computer Science Laboratory, 1999, `http://www.erlang.org/download/erl_spec47.ps.gz`.

[136] B. Dawson, Comparing floating point numbers, `http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm`, 2008.

[137] T. Arts, L. M. Castro, and J. Hughes, Testing Erlang data types with Quviq QuickCheck, in *Proceedings of 7th ACM SIGPLAN Workshop on Erlang*, pages 1–8, ACM Press, 2008.

[138] R. Osherove, *The Art of Unit Testing*, Manning Publications Co., 2009.

[139] T. Haerder and A. Reuter, Principles of transaction-oriented database recovery, ACM Computing Survey **15**(4), 287–317 (1983).

[140] E. F. Codd, A relational model of data for large shared data banks, Communications of the ACM **13**(6), 377–387 (1970).

[141] M. Bajec and M. Krisper, A methodology and tool support for managing business rules in organisations, Information Systems **30**(6), 423–443 (2005).

[142] J. Dietrich and A. Paschke, On the test-driven development and validation of business rules, in *Proceedings of 4th International Conference on Information Systems Technology and its Applications*, volume 63, pages 31–48, 2005.

[143] D. Willmor and S. M. Embury, Testing the Implementation of Business Rules Using Intensional Database Tests, in *Proceedings of Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 115–126, IEEE Computer Society Press, 2006.

[144] JUnit: Testing framework for Java, http://www.junit.org, 2008.

[145] N. Paladi and T. Arts, Model based testing of data constraints: testing the business logic of a Mnesia application with Quviq QuickCheck, in *Proceedings of 8th ACM SIGPLAN Workshop on Erlang*, pages 71–82, ACM Press, 2009.

[146] V. Chanana and A. Koronios, Data Quality Through Business Rules, in *Proceedings of International Conference on Information and Communication Technology*, pages 262–265, Mar. 2007.

[147] A. I. Standard, Database Language SQL, http://www.cse.iitb.ac.in/dbms/Data/Papers-Other/SQL1999/ansi-iso-9075-2-1999.pdf, 1999.

[148] T. Demarco, Software Engineering: An Idea Whose Time Has Come and Gone?, IEEE Software **26**(4), 95–96 (2009).

[149] R. R. Loka, Software Development: What Is the Problem?, Computer **40**(2), 110–112 (2007).

[150] L. M. Castro, V. M. Gulías, C. Abalde, and J. París, Environment-independent methodology for accessing external data sources, WSEAS Transactions on Information Science and Applications **5**(9), 1–10 (2008).

[151] L. M. Castro, V. M. Gulías, C. Abalde, and J. París, Database access and patterns in Erlang/OTP, in *Proceedings of 8th Conference on Applied Informatics and Communications*, pages 73–78, World Scientific and Engineering Academy and Society (WSEAS), 2008.

[152] C. Abalde, V. M. Gulías, L. M. Castro, C.Varela, and J. S. Jorge, Erlang/OTP Framework for Complex Management Applications Development, in *Proceedings of 3rd International Conference on Web Information Systems and Technologies*, pages 422–425, INSTICC Press, Mar. 2007.

[153] L. M. Castro, M. A. Francisco, and V. M. Gulías, A Practical Methodology for Integration Testing, Lecture Notes in Computer Science **5717**, 881–888 (2009).

[154] L. M. Castro, M. A. Francisco, and V. M. Gulías, Testing Integration of Applications with QuickCheck, in *Proceedings of 12th Twelve International Conference on Computer Aided Systems Theory*, pages 299–300, Feb. 2009.

[155] L. M. Castro, V. M. Gulías, and M. A. Francisco, Applications integration: a testing experience, in *Proceedings of 6th Workshop on System Testing and Validation*, Dec. 2008.

[156] M. Armstrong, *Management Processes and Functions*, Management studies, Institute of Personnel and Development, 1990.

[157] Who uses Erlang for product development?, `http://erlang.org/faq/introduction.html#1.5`, 2010.

[158] Haskell in Industry, `http://www.haskell.org/haskellwiki/Haskell_in_industry#Haskell_in_Industry`, 2010.

[159] TIOBE Software BV, Programming Community Index, `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, 2010.

[160] Crain Communications Inc., Largest risk management information systems providers, Business Insurance (Apr. 2007).