Requirement Analysis Evolution through Patterns

Luca Vetti Tagliati *LucaVT@gmail.com*
Roger Johnson *rgj@dcs.bbk.ac.uk*
George Roussos *gr@dcs.bbk.ac.uk*
Computer Science and Information Systems
Birkbeck University of London

Abstract

This paper presents a strategy, based on requirement patterns (RP), aimed at improving the requirement analysis discipline by allowing business analysts (BA) to produce more reliable SW requirements in a significantly shorter time, minimising the overall requirement risks. In numerous business organisations, IT systems are increasing their strategic significance. In extremely competitive environments, such as investment banking -where this methodology has been tested- modern and advanced IT systems can enable the organisation to obtain and to maintain a predominant position in the market, which in turn results in a greater ROI.

Regrettably a number of academic and industrial studies depict a catastrophic picture about SW projects: most of them are likely to fail and, logically, the probability of failure grows with the size of the project. The project failure factor varies within a range of 50% - 70%. Furthermore, such studies clearly show that requirements is the area where the major risks reside. The proposed strategy is based on the introduction of elegant, well-proven, technology-agnostic, architecturally-compatible, simple and reusable patterns that, focusing on the functional requirements, expand on other requirement analysis artefacts such as domain object model (DOM), business rules (BR), user interface (UI) and glossary.

Keywords: requirement patterns, functional requirements, non-functional requirements, domain object model, business rules, use cases, UML.

1. Introduction

This paper considers the Use Case (UC) formalism as the foundation of the requirement analysis discipline. However, it also provides practitioners with a number of other requirement-related artefacts that are beneficial to BAs regardless of the process and formalism employed.

The RP core consists of UC models, including their specifications, which are linked to other artefacts like DOM, UI design, BR and glossary. Furthermore, this paper demonstrate the opportunity to associate the RPs with artefacts belonging to other models, like system test cases (see Fig. 1). According to Ross Collard [1], UCs and test cases make an effective combination in two ways: when the UCs are complete, accurate and clear, the process of deriving the test case is mechanical. If the UCs are not in good shape, deriving test cases facilitate debugging the UCs. Therefore, while UCs describe in detail the services that the system will have to deliver, the test cases ensure that the system provides these services as agreed.

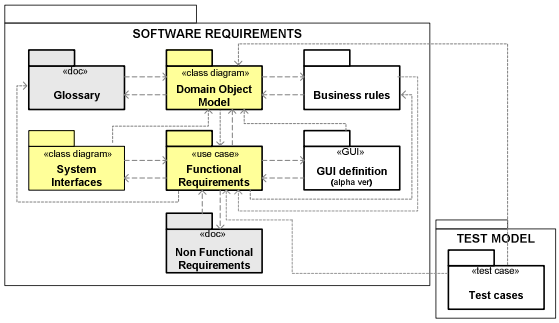

Figure 1. Requirement models relationships

2. Rationale

The requirements patterns concept proposed by this research focuses on the functional requirements modelled through the UC notation as described in the OMG UML specification [2]. This is central to all other artefacts. The behaviour of each UC included in the corresponding diagrams is given in terms of a structured natural language i.e. a template. This allows practitioners to illustrate the sequence of interactions between actors and the systems necessary to achieve the UC goal. Restricting the functional requirements modelling exclusively to UC diagrams and the corresponding specifications would be too rigid not only for the requirements patterns concept but also for ordinary UC models. Therefore, it is necessary to design a mechanism to parameterise the UCs and requirement models to enable their convenient re-use.

This mechanism, consistently with the UC notation, has to:

- present a variable level of formality (it is important to remember that one fundamental audience of UC models is the user community);
- be organised in a core part that cannot be easily changed plus a number of parametric sections whose definition represents the customisation of each single pattern to the specific need.

The solution envisages using the BR document to delegate the definition of the customisable parts, which are directives that differentiate the use of specific requirements. In this way it is possible to define UC scenarios with parametrical sections whose specification is delegated to well-defined entries (paragraphs) included in the business rules document. Therefore, the traditional use of BR is enhanced to include the specification of the parameterisable behaviour.

The adoption of this technique to model UCs presents a number of advantages, independently from the usage of the requirements patterns. For example, it reduces redundancy. Typically, the same BR are referred to by several UCs and by other artefacts (e.g. design model). Therefore, instead of copying and pasting the same BR across a number of different artefacts, with evident problems related to maintainability and traceability, it is possible to refer to the same one stored in the BR document. Furthermore, BR modelling, depending on their nature (constraints, algorithm, etc.), can require different notations. For example, one of the most effective ways of expressing an algorithm is to use the UML activity diagram notation, while some market regulations are better expressed in natural language. Therefore, UC notation is not always the most appropriate tool to express BRs. For the above-mentioned reasons, BRs must be stored in a single artefact and then be referenced by all the others.

The main mechanism for customising the proposed RPs, referred to as a light-weight customisation, consists of specifying the content of the BRs referred to by the requirements themselves or simply accepting the proposed ones. However, this is not the only way as, in fact, it is also possible to change everything else including the UC specification itself (this is a heavy-weight customisation). However, these kinds of changes are pervasive and therefore they should be used only when it is absolutely necessary.

Another advantage of this technique is related to its capability to distinguish, clearly from the source, the part of the requirements that do not change often from the ones that vary more frequently (i.e. sections defined in the business rules document). This should provide development teams with an important input for the design and implementation of reusable business components.

RPs focus on UCs and expand on other models like DOM, UI and glossary. Therefore, when a pattern that provides the solution for a specific requirement refers to well-defined, interrelated business entities (i.e. a portion of the DOM) and/or a UI model, these can also be incorporated in the corresponding requirements model. Furthermore, it is possible to include in the glossary an explanation, given in natural language, of the concepts presented in the class diagrams.

A typical approach for business requirements analysis and documentation consists of focusing first on the services that the system will have to deliver, modelled via the corresponding UCs, and then validating them considering the organisation of the corresponding business entities, modelled via class diagrams. Typically, the definition of these class diagrams requires a review of the UCs. This is the case, for example, where the business entity structure initially assumed in the UC was not fully correct or complete.

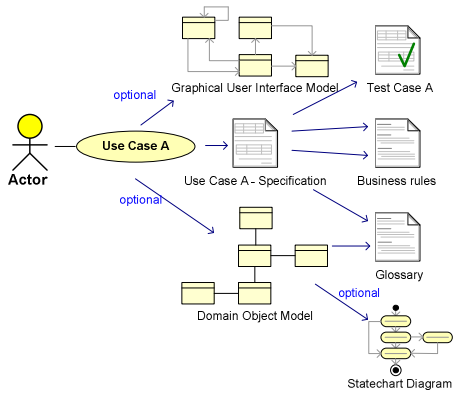

Figure 2. Models of requirement patterns

The DOM is a key artefact for SW development not only in balancing the related UCs, but also in using as an input for the production of other fundamental artefacts like: components design, database design, system messages design and user interfaces. Furthermore, the analysis of a number of business services is better approached using the opposite strategy: defining the business entities' organisation and then consequently modelling the UCs that, manipulating these entities, are able to provide users with requested services. RPs also fully supported this approach: BAs can decide to include in their models specific section of a DOM and then design their own UCs. Therefore, although the RPs idea focuses on UCs, the other models also assume a significant relevance (Figure 2).

The overall RP idea consists of enabling BAs to search through RP collections for specific

issues/domain problems and to extract the model required. Each pattern can comprise a number of models:

- one or more UC diagrams;
- a set of UC specifications (templates) which specify the dynamics of each UC present in the diagram mentioned above;
- a number of pre-defined paragraphs to be included in the BRs document whose definition represents the main mechanism to customize the RP. Furthermore, UC specifications can contain topics to be added into the overall glossary;
- a class diagram which models the business entities, including their relationships, referred to by the UC. A textual description of the mentioned entities can be included in the glossary. Some business entities can present a well-defined lifecycle modelled by a corresponding UML statechart diagram which can be included in the RP as well;
- an optional class diagram which models the UI structure including the navigation associated with the services described in the UC;
- a few test cases that describe the test to be performed to verify that the implementation of the specified services are correct and robust.

3. Case study

The following paragraphs discuss a small portion of a case study in order to provide readers with the practical aspect of the proposed theory. In particular, this methodology has been successfully employed in a global investment bank for the development of a security system designed to implement authentication, authorisation and data privacy services. The experiment employed patterns previously designed and extracted from requirements successfully used for similar projects. These included twenty eight UCs, a large DOM, an extensive BRs document, etc. From this large pattern collection, we have selected an example which is related to what is commonly and **incorrectly** perceived to be a simple service: user authentication. This service was selected because: everybody is familiar with it, it allows the presentation of a pattern that includes a number of different diagrams, it is a service that everybody initially would consider extremely simple and straightforward, but a more detailed analysis highlights a number of important aspects that not everybody would think about. In the employed approach, BAs would start from the UC diagram depicted in figure 3.

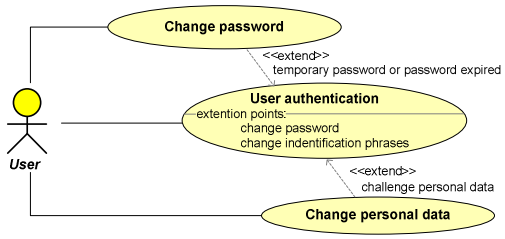

Figure 3. Authentication use case diagram

For each of the presented UCs, the RP, include the corresponding specification. In this paper, due to space limitations, only a small fragment of first one (User Authentication) is presented (see fig. 4).

USE CASE UC: SEC.AUTHENT		**User authentication**	**Date:**	29/Aug/2005
			Version:	0.00.001
Description:		The Log-in service allows users to gain access to the system. The system verifies the credentials inserted by the user and, if these are valid, then the user is authenticated, otherwise the system executes a well-define procedure depending on the number of consecutive failed log-in attempts. The authentication process is only a pre-condition for the execution of the sensitive system's services: authenticated users have to be also authorised.		
User priority:		Medium		
Primary actor:		**User.** This is a generic user (abstract). His/her interest in this use case is to log into the system.		
Preconditions:		The system is available.		
Post-conditions on success:		The system authenticates the user.		
Post-conditions on failure:		The system refuses access to the user and it performs the corresponding management actions.		
Trigger:		The user requests to log into the system.		
MAIN SCENARIO				
1	System:	Displays the initial "log-in" screen. **UI:** SEC::LOGIN		
2	User:	Specifies the requested credentials. **BR:** SEC::login_credentials		
3	System:	Determines that the user login is valid. **BR:** SEC::login_validation		
4	System:	Determines that the user status is valid. **BR:** SEC:: login_user_status_validation		
5	System:	Verifies that the inserted password matches the corresponding internal one. **BR:** SEC:: password_matching		
6	System:	Verifies that the user's password is in a valid status. **BR:** SEC::password_status_valdation		
7	System:	Determines that the same credentials are not currently in use. **BR:** SEC::credentials_not_in_use		
8	System:	Resets the consecutive unsuccessful logins counter.		
9	System:	Logs the login action into the security audit trail.		
10	System:	Loads the user's profile.		
11	System:	Verifies that the user's profile is valid. **BR: SEC::**user_profile_status_validation		
12	System:	Insert the user' login in the "users current logged in" list. **DOM:** LoggedIn.users.add(current_user)		
13	System:	Shows the user's menu.		
14	System:	The use case ends.		
Alternative Scenario:		**User does not specify the credentials.**		
3.1	System:	Shows an error message.		
3.2	System:	Resumes at point 1.		
Alternative Scenario:		**User login not valid.**		
3.1	System:	Determines that the maximum number of consecutive failed attempts from the same connection has not been reached. **BR:** SEC::maximum_attempts_connection		
3.2	System:	Increases the number of consecutive failed attempts associated with the connection.		
3.3	System:	Shows an error message.		
3.4	System:	Logs the login action into the security audit trail.		
3.5	System:	Resumes at point 1.		
Alternative Scenario:		**Specified password does not match the internal one.**		
5.1	System:	Determines that the maximum number of consecutive failed attempts related to the user has not been reached. **BR:** SEC:: maximum_attempts_against_user		
5.2	System:	Increases the number of consecutive failed attempts associated with the user and the ones associate with the connection.		
5.3	System:	Shows an error message.		
5.4	System:	Logs the login action into the security audit trail.		
5.5	System:	Resumes at point 1.		

Figure 4. Authentication UC specification (Main and some alternative scenarios)

The proposed version is particularly appropriate for enterprise systems. This is because it includes the logic necessary to detect possible intrusion attempts (from a specific location and/or against a precise user) and to check that other users are not currently logged-in with the same credentials. Furthermore, there are extensive controls related to the status of the user,

his/her profile and password, which are complex objects with a well-defined cycle of life, etc. The authentication UC specification presents a number of areas where the corresponding default behaviour (i.e. business logic) can be redefined. The definition of the corresponding BRs is the main lightweight mechanism to customize the UC specification. In particular, the proposed UC specification presents the following business rules: login_credentials, login_validation, login_user_status_validation, password_matching, password_status_valdation, credentials_not_in_use, user_profile_status_validation,maximum_attempts_connection, maximum_attempts_against_user, maximum_attempts_against_user. These allow practitioners to define simple behaviour, like the maximum number of consecutive failed attempts allowed from the same remote address, or more complex ones like the conditions that, if verified, force the user to change his/her password.

The default rule states that the user's password has to be changed if its status is temporary, which means that the password has been automatically issued by the system, or its validity time window has expired. These are examples of parts of the service that are subject to change from one implementation to another, and therefore their implementation requires a degree of flexible. Other UCs foresee more complex business rules whose definition is given in term of algorithm modelled by UML activity diagrams, like the calculation necessary to generate unique user id.

The authentication UC specification also includes a number of references to the pre-defined corresponding part of the DOM (figure 5), highlighted by the string "DOM" written in bold. As mentioned before, UCs and DOM describe two different projections of the same "entity", which, in this case, is the authentication service requirement.

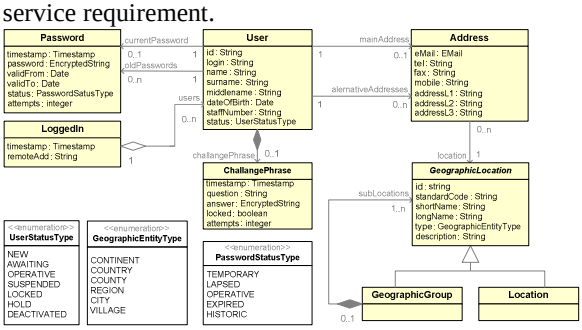

Figure 5. Part of the DOM

The BA, as usual, can decide to integrate the propose sections of the DOM referenced by the UC specification, or to use his/her own. In this case, the most significant entities: user, profile and password, are provided with the corresponding UML statechart diagrams that can be included in the BA model.

Moreover, the UC specification refers to a well-defined user interface (UI:SEC::USER_AUTH) whose object oriented model is a part of the user requirement as well.

Finally, the pattern adds in a number of terms (for example, *Authentication* and *Authorisation*), including the corresponding definition, which can be included in the Glossary document.

4. Requirements patterns categories

The concept of pattern in the SW community has been used with a number of different meanings. The pattern notion considered by this paper presents a high level of compatibility with the original Alexandrian idea [12] that each pattern describes a recurring problem in the particular problem domain including the corresponding solution. This provides practitioners with the possibility of reusing the solutions a number of times without having to study the same problem over and over again [3]. Therefore, the real core idea is to produce a catalogue of elegant, well-proven, extensible and re-usable requirements patterns in the same way that the authors of the book [4] did for the design model. In particular, RPs are reusable, well-proven, architecture friendly and high-quality requirement models for recurrent problems, obtained as result of the experiences from development of real projects. These patterns are provided with the context of their usage, including forces, and they are designed to be customisable by modifying the linked business rules.

From the analysis of real world projects requirements, it has been possible to divide RPs into two main categories: domain specific and general purposes. The former are particularly suitable for specific domains like security, e-commerce, banking, etc. This category presents some similarities with the work of Bjørner [5], related to his studies to formally define the problem domain via a formal mathematical language. The latter are patterns, like data entry, searches, data analysis, and so on. These are typically extracted as a result of the process of reengineering patterns belonging to the previous category. Therefore, the previous category is a first-level application of the patterns present in this set. However, both categories are proper patterns since they provide a well-defined solution to recurrent problems, either domain specific or more general.

RPs can provide practitioners with:

- elegant solutions that not everybody would think of immediately;
- technology and programming language agnostic;
- architecturally compatible and consistent solutions that have been proved through successful implementation in other projects;

- well-proven solutions identified through the analysis of real projects;
- high level of flexibility;
- simple but effective solutions;
- reusable solutions;
- a framework for developing CASE tools;
- a set of superior solutions that can also be very useful for training purposes.

5. Advantages

The RPs adoption produced the following advantages:

time saving. These RPs allowed BA to save time and effort invested in modelling the requirements and therefore they were able to invest more time in the proper requirements analysis and less in the formal aspect of their modelling. This time saved is not only related to the initial production, but it is extended to the number of re-factoring iterations that BAs typically undertake. Often reviewing a model generates a ripple of a number of other models. For example, reviewing a package in the DOM necessitates the review of all UCs that, starting from the described entities, generate a number of services, the UI, the business rules, etc.

higher-level of quality. Analysis gathering discipline proved to be a complicated and particularly critical part of the SW development process. It is not always possible to think ahead about all the different aspects of a requirement (especially for the more complex ones) and as a result, a number of changes can occur that can produce serious consequences of the process outcome (e.g. requirement creep). Furthermore, it is not always possible or affordable to employ a BAs expert in very specific domains, like e.g. in IT security. Therefore, RPs are extremely convenient in these scenarios. Furthermore, RPs explore all scenarios and possible alternatives present in the analysed topic and therefore they do not leave any aspect unexplored, often they present a way of modelling the same requirement that not everybody would think of immediately, they are "implementation friendly" and consistence since they have been identified in previous projects, etc. Finally, each pattern, typically, includes other models like the DOM and the UI model that are often neglected because of a project's time and budget limitations. As proof, the issues tracking system (this project used the software Jira) showed that there was not a single log related to change requirements for the security system. They were all related to fixing and only 10% to the implementation of new services.

risk reduction. This advantage is a direct consequence of the overall quality enhancement described above. Furthermore, since the RPs are well-proven solutions identified through the analysis of real-projects, their feasibility and their ease of implementation are guaranteed. In addition, these patterns provide BAs with important tools for verifying the validity, accuracy and completeness of the requirements specified by users;

time and cost saving. These objectives are the logical consequence of a number of factors. First of all, BAs did not have to model a number of requirements since these were already provided by the patterns. UCs and scenarios can be labour-intensive to capture and document. ([10] and [11]). Furthermore, the requirement models present a high quality level and they are architecture-friendly. The requirements patterns can be raised to a further level by including design model and implementation. In fact, other outputs of the implementation of the security system are reusable design model related to the security RPs. Therefore, the selection of a RP has the potentiality to bring with it models belonging to the design and implementation phases. Finally, their presence allows managers to organize teams where not all business analysts need to be experienced.

standardisation of the business areas. A number of practitioners started realising that the large availability of out-of-the-box components evocated by the *.Net* and *J2EE* architecture has not happened. One of the explanatory factors can be found in the lack of business domains standardisation. This problem can be solved with the RPs, which provide a core with the description of the flows of actions, including the point where the behaviour can vary. Therefore, this should provide development teams with a standardisation that would allow them to produce well-defined and reusable software components. As proof, the development team is investigating the idea of releasing a few icomponents to the open source community.

learning. RPs provide junior BAs with an effective way of improving their technique. Furthermore, given their quality and elegance they allow the less experienced analysis to produce high quality outcome.

6. Related work

The RPs idea is in some ways related to previous works in this area. The most relevant works are:

parameterised UCs introduced by Cockburn [7], where two examples of patterns are discussed. One such pattern, the *"find whatever"*, represents the researching data function, and the second relates to a typical CRUD functionality.

"Patterns for Effective Use Cases" [8], in this case there are differences starting from the patterns notion, which is clearly illustrated by several quotes included in the book. E.g. they propose to consider patterns as

merely a sign of quality, and strategy. They do not consider pattern language as a complete strategy for writing requirements, but as a set of guidelines to support practitioners fill a gap in their knowledge, evaluate UCs quality, etc. Therefore, there is an important divergence from the idea presented in this paper.

"**Use Cases Patterns and Blueprints**" by G. Övergaard and K. Palmkvist [9]

Bjørner's study [5] where the author investigates specific domains (like the railways) with the aim of representing them via a formal mathematical language

The current IT body of knowledge embraces a number of patterns methodologies applied to other disciplines of the software development process, like analysis patterns and design patterns. Although these are extremely interesting, they are out of the scope of this paper.

Although several academic studies and empirical researches present some similarities with this approach, there are also a number of important differences. The most relevant ones shared by all other approaches are:

- some approaches do not consider UCs at all (e.g. [5], [8])
- approaches that focus on UCs are often not fully compliant with the corresponding standard ([7])
- most of the approaches focus on one artefact and do not expand to the wider concept of SW requirements. Either they focus on the functional requirements or on a sort of static view ([5]). Other important requirements artefacts, like DOM and UI, are simply ignored
- only one approach ([9]) tries to makes use of BR but not in a way that would promote reusability
- no single approach includes specific mechanisms for a convenient re-use and customisation of RPs.

Övergaard and Palmkvist ([9]) propose several UC patterns based on a high level of conceptuality that poses a number of problems for their re-use in real projects. E.g., as a matter of comparison, it is possible to analyse their version of the user authentication UC, called log-in. This is unexpectedly integrated with the logoff UC (the same UC encapsulates two completely different and logically opposite services). From the analysis of this UC it is possible to highlight that it is not considered the possibility of fraudulent security attacks, there is not a scenario aimed at locking a user account in case the maximum number of consecutively failed attempts to login has been reached, there are no further checks on the password data, etc. Therefore the reuse of their patterns it is not straightforward. It will likely require the production of a further and more detailed version of the proposed UCs. Finally, the UCs notation is adopted in an unconventional fashion highlighted by the unusual presence of two main scenarios.

7. Conclusion

The overall hypothesis is that the RPs strategy provides practitioners with an effective instrument to produce higher quality requirements analysis more efficiently. This, in turn, produces two major advantages:

project cost reduction: requirements are gathered more rapidly, there are fewer change requirements, etc;

risks reduction. This is achieved because the extracted requirements present a higher-level quality and because the saved time can be invested in more critical activities.

The latter advantage is particularly important since commercial surveys still indicate that the major number of software projects fail because of problems with the requirements stage.

The initial study and corresponding investigation showed a huge success during the requirement phases where UCs and the corresponding DOM were produced by copying patterns from a document. The whole set was produced in only twenty three man days and with virtually no change was requested during the whole process. Furthermore, architects could benefit straightaway from a whole set of requirements that allowed them to design the architecture and the system with no delays.

10. References

[1] R. Collard – "Test Design, Software Testing & Quality Engineering" – July 1999

[2] OMG UML 2 specification

[3] C. Alexander, "A Pattern Language", New York: Oxford University Press, 1977

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides – "Design patterns, Elements of Reusable Object-Oriented Software" – Addison Wesley, 1994

[5] D. Bjørner: A Cloverleaf of Software Engineering, IEEE SEFM'05

[6] C. Wohlin, K. Henningsson, M. Höst , "Empirical Research Methods in Software Engineering", 1998

[7] A. Cockburn, "Writing effective use cases", Addison-Wesley, September 2000

[8] S. Adolph, P. Bramble, A. Cockburn, A. Pols, in the book "Patterns for Effective Use Cases" Addison-Wesley, 08/2002

[9] G. Övergaard and K. Palmkvist – "Use Cases Patterns and Blueprints" – Addison Wesley – Nov/2004

[10] P.A. Gough, F.T. Fodemski, S.A. Higgings, and S.J. Ray "Scenarios – An Industrial Case Study and Hypermedia Enhancements" Proc. Second IEEE Symposium Requirements Engineering. IEEE Computer Society, pages 10-17. 1995

[11] T. Royer, "Using Scenario-Based Designs to Review User Interface Changes and Enhancements", proc. DIS95: Designing Interactive Systems, Ann Arbor. Pages 236-246 – 1995

[12] C. Alexander, "A Pattern Language", New York: Oxford University Press, 1977