2 Anticipating Change in Requirements Engineering

Soo Ling Lim and Anthony Finkelstein

Department of Computer Science, University College London
s.lim@cs.ucl.ac.uk and a.finkelstein@cs.ucl.ac.uk

Abstract Requirements change is inevitable in the development and maintenance of software systems. One way to reduce the adverse impact of change is by anticipating change during requirements elicitation, so that software architecture components that are affected by the change are loosely coupled with the rest of the system. This chapter proposes Change-oriented Requirements Engineering (CoRE), a method to anticipate change by separating requirements into layers that change at relatively different rates. From the most stable to the most volatile, the layers are: patterns, functional constraints, non-functional constraints, and business policies and rules. CoRE is empirically evaluated by applying it to a large-scale software system, and then studying the requirements change from development to maintenance. Results show that CoRE accurately anticipates the relative volatility of the requirements.

Keywords: Requirements change management; shearing layers; goal modelling.

2.1 Introduction

Requirements change is inevitable in the development and maintenance of software systems. As the environment and stakeholders' needs continuously change, so must the system, in order to continue meeting its intended purpose. Studies show that requirements change accounts for a large part of the rework in development, in some cases up to 85% [4, 16, 34]. As such, it is one of the top causes of project failure [30], and often regarded as one of the most chronic problems in software development [14].

One way to reduce the impact of change is to anticipate change during requirements elicitation. Identifying volatile requirements from the start enables the design of the system such that architectural components that realise the requirements are loosely coupled with the rest of the system [29, 33]. Then, software changes to accommodate the requirements change are easier to implement, reducing the amount of rework.

Despite the centrality of change management in requirements engineering, the area of change management lacks research [33]. Existing requirements engineering methods regard change anticipation as a separate activity *after* documentation [29]. Without the notion of future changes, the documentation mixes stable and volatile requirements, as the existing method section will show. Existing change anticipation approaches are guidelines that rely on domain experts and experienced requirements engineers [29], who may be absent in some projects. Finally, existing literature is almost entirely qualitative: there is no empirical study on the accuracy of these guidelines in real projects.

To address these problems, this chapter proposes Change-oriented Requirements Engineering (CoRE), an expert independent method to anticipate requirements change. CoRE separates requirements into layers that change at relatively different rates *during* requirements documentation. This informs architecture to separate components that realise volatile requirements from components that realise stable requirements. By doing so, software design and implementation prepares for change, thus minimising the disruptive effect of changing requirements to the architecture.

CoRE is empirically evaluated on its accuracy in anticipating requirements change, by first applying it to the access control system project at University College London, and then studying the number of requirements changes in each layer and the rate of change over a period of 3.5 years, from development to maintenance. This study is one of the first empirical studies of requirements change over a system's lifecycle. The results show that CoRE accurately anticipates the relative volatility of the requirements.

The rest of the chapter is organised as follows. Section 2.2 reviews existing methods in requirements elicitation and change anticipation. Section 2.3 introduces the idea behind CoRE. Section 2.4 describes CoRE and Section 2.5 evaluates it on a real software project. Section 2.6 discusses the limitations of the study before concluding.

2.2 Existing Methods

Requirements Elicitation

In requirements elicitation, model-based techniques, such as *use case* and *goal modelling*, use a specific model to structure their requirements, which often mixes stable and volatile requirements.

Use case is one of the common practices for capturing the required behaviour of a system [13, 7]. It models requirements as a sequence of interactions between

the system and the stakeholders or other systems, in relation to a particular goal. As such, a use case can contain both stable and volatile requirements. An example use case for an access control system is illustrated in Table 2.1. In the use case, "displaying staff member details" in Step 1 is more stable than "sending photos to the staff system" in Step 4, because the verification and retrieval of person information is central to all access control systems, but providing photos to another system is specific to that particular access control system.

Table 2.1. Use Case: Issue Cards to Staff

Step	Action Description
1.	The card issuer validates the staff member's identity and enters the identity into the system.
2.	The system displays the staff member's details.
3.	The card issuer captures the staff member's photo.
4.	The system generates the access card and sends the photo to the staff system.

Goal modelling (e.g., KAOS [32] and GBRAM [1]) captures the intent of the system as goals, which are incrementally refined into a goal-subgoal structure. High-level goals are, in general, more stable than lower-level ones [33]. Nevertheless, goals at the same level can have different volatility. For example, the goal "to maintain authorised access" can be refined into two subgoals: "to verify cardholder access rights" and "to match cardholder appearance with digital photo." The second subgoal is more volatile than the first as it is a capability required only in some access control systems.

Requirements Documentation

When it comes to documenting requirements, most projects follow standard requirements templates. For example, the *Volere Requirement Specification Templat*e by Robertson and Robertson [24] organises requirements into functional and non-functional requirements, design constraints, and project constraints, drivers, and issues. The example access control system has the functional requirement "the access control system shall update person records on an hourly basis." Within this requirement, recording cardholder information is more stable than the frequency of updates, which can be changed from an hour to 5 minutes when the organisation requires its data to be more up-to-date.

A standard requirements template is the *IEEE Recommended Practice for Software Requirements Specification* [11]. The template provides various options (e.g., system mode, user class, object, feature) to organise requirements for different types of systems. For example, the system mode option is for systems that behave differently depending on mode of operation (e.g., training, normal, and em-

ergency), and the user class option is for systems that provide different functions to different user classes. Nevertheless, none of these options organise requirements by their volatility, which is useful for systems with volatile requirements, such as software systems in the business domain [14].

Requirements Change Management

In requirements change management, one of the earliest approaches by Harker *et al*. [10] classifies requirements into *enduring requirements* and *volatile requirements*. *Volatile* requirements include *mutable requirements*, *emergent requirements*, *consequential requirements*, *adaptive requirements*, and *migration requirements*. Harker *et al*.'s classification is adopted in later work by Sommerville, Kotonya, and Sawyer [15, 28, 29]. Although the classification clearly separates stable requirements from volatile ones, the relative volatility among the types of volatile requirements is unknown. For example, *mutable requirements* change following the environment in which the system is operating and *emergent requirements* emerge as the system is designed and implemented. Without further guidelines, it is unclear whether a mutable requirement is more volatile than an emergent requirement.

Later work provides guidelines to anticipate requirements change. Guidelines from van Lamsweerde include: (1) stable features can be found in any contraction, extension, and variation of the system; (2) assumptions and technology constraints are more volatile than high-level goals; (3) non-functional constraints are more volatile than functional requirements; and (4) requirements that come from decisions among multiple options are more volatile [33]. Guidelines from Sommerville and Sawyer [29] are: (1) identify requirements that set out desirable or essential properties of the system because many different parts of the system may be affected by the change; and (2) maintain a list of the most volatile requirements, and if possible, predict likely changes to these requirements. The caveat with these guidelines is that they require experienced requirements engineers or domain experts to identify requirements that are likely to be volatile, and still, errors can occur [29].

To summarise, existing requirements elicitation methods lack the ability to anticipate change. In addition, existing requirements templates do not separate requirements by their volatility, and existing change management approaches are expert dependent. In contrast, in the CoRE method proposed in this chapter, requirements anticipation is part of the requirements modelling process and independent of the person doing the analysis.

2.3 The Shearing Layers

CoRE adopts the concept of shearing layers from building architecture. This concept was created by British architect Frank Duffy who refers to buildings as composed of several layers of change[1] [5]. The layers, from the most stable to most volatile, are site, structure, skin, services, space plan, and "stuff" or furniture (Fig. 2.1). For example, services (the wiring, plumbing, and heating) evolve faster than skin (the exterior surface), which evolves faster than structure (the foundation). The concept was elaborated by Brand [5], who observed that buildings that are more adaptable to change allow the "slippage" of layers, such that faster layers are not obstructed by slower ones. The concept is simple: designers avoid building furniture into the walls because they expect tenants to move and change furniture frequently. They also avoid solving a five-minute problem with a fifty-year solution, and vice versa.

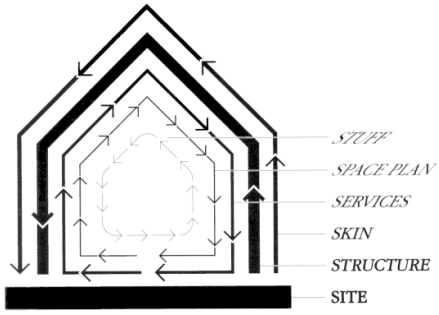


Fig. 2.1. The shearing layers of architecture [5].

The shearing layer concept is based on the work of ecologists [22] and systems theorists [26] that some processes in nature operate in different timescales and as a result there is little or no exchange of energy or mass or information between them. The concept has already been adopted in various areas in software engineering. In software architecture, Foote and Yoder [9], and Mens and Galal [20] factored artefacts that change at similar rates together. In human computer interaction, Papantoniou *et al*. [23] proposed using the shearing layers to support evolving design. In information systems design, Simmonds and Ing [27] proposed using rate of change as the primary criteria for the separation of concerns.

Similar to the elements of a building, some requirements are more likely to change; others are more likely to remain the same over time. The idea behind

[1] http://www.predesign.org/shearing.html

CoRE is to separate requirements into shearing layers, with a clear demarcation between parts that should change at different rates.

2.4 Change-oriented Requirements Engineering (CoRE)

The Shearing Layers of Requirements

CoRE separates requirements into four layers of different volatility and cause of change. From the most stable to the most volatile, the layers are: patterns, functional constraints, non-functional constraints, and business policies and rules (Fig. 2.2). Knowledge about patterns and functional constraints can help design and implement the system such that non-functional constraints, business policies and rules can be changed without affecting the rest of the system.

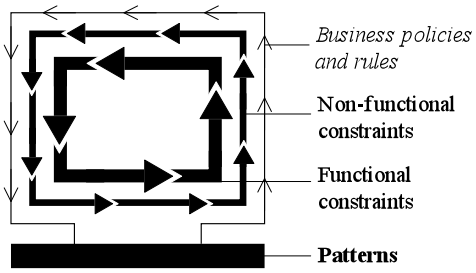


Fig. 2.2. The shearing layers of requirements.
The layers with more arrows are more volatile.

Patterns. A pattern is the largest combined essential functionality in any variation of a software component that achieves the same goal. As such, they remain unchanged over time unless the goal is no longer needed. For example, the goal of an inventory system is to maintain a stock of items for sale. It can be achieved by the `Inventory` pattern illustrated in Fig. 2.3 (a) with functionalities such as making reservations, adding and finding products. These functionalities have existed long before software systems and are likely to remain unchanged. Different patterns can be found in different domains, e.g., patterns in the medical domain revolve around patients, doctors, patient records [28] and patterns in the business domain revolve around products, customers, inventory [2]. In the business domain, Arlow

and Neustadt [2] developed a set of patterns which they named enterprise archetypes as the patterns are universal and pervasive in enterprises[2]. Their catalogue of patterns consists of various business related pattern, including the ones in Fig. 2.3.

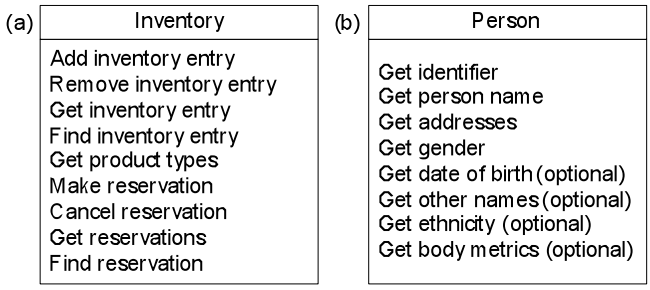


Fig. 2.3. Example patterns: (a) `Inventory` pattern (b) `Person` pattern [2].

Functional constraints. Patterns allow freedom for different instantiations of software components achieving the same goals. In contrast, functional constraints are specific requirements on the behaviour of the system that limit the acceptable instantiations. These constraints are needed to support the stakeholders in their tasks, hence remain unchanged unless the stakeholders change their way of working. For example, an access control system's main goal is to provide access control. The pattern assigned to this goal is the `PartyAuthentication` archetype that represents an agreed and trusted way to confirm that a party is who they say they are [2]. A functional constraint on the achievement of this goal is that the system must display the digital photo of the cardholder when the card is scanned, in order to allow security guards to do visual checks.

Non-functional constraints. A non-functional constraint is a restriction on the quality characteristics of the software component, such as its usability, and reliability [6]. For example, the ISO/IEC Software Product Quality standard [12] identifies non-functional constraints as a set of characteristics (e.g., reliability) with sub-characteristics (e.g., maturity, fault tolerance) and their measurable criteria (e.g., mean time between failures). Changes in non-functional constraints are independent of the functionality of the system and occur when the component can no longer meet increasing quality expectation. For example, in an access control system, a person's information has to be up-to-date within an hour of modification. The constraint remains unchanged until the system can no longer support the increasing student load, and a faster service is needed.

[2] From this point on, the word "archetype" is used when referring specifically to the patterns by Arlow and Neustadt [2].

Business policies and rules. A business policy is an instruction that provides broad governance or guidance to the enterprise [3, 21]. A business rule is a specific implementation of the business policies [3, 21]. Policies and rules are an essential source of requirements specific to the enterprise the system operates in [25, 3]. They are the most volatile [3], as they are related to how the enterprise decides to react to changes in the environment [21]. For example, a university deploying the access control system may have the business policy: *access rights for students should correspond to their course duration*. The business rule based on the policy is: *a student's access rights should expire 6 months after their expected course end date*. For better security, the expiration date can be shortened from 6 months to 3 months after the students' course end dates.

CoRE Method

CoRE is based on goal modelling methods [8, 35]. To separate requirements into the shearing layers, CoRE applies five steps to the goals elicited from stakeholders (Fig. 2.4). The access control system example for a university is used as a running example to demonstrate the method.

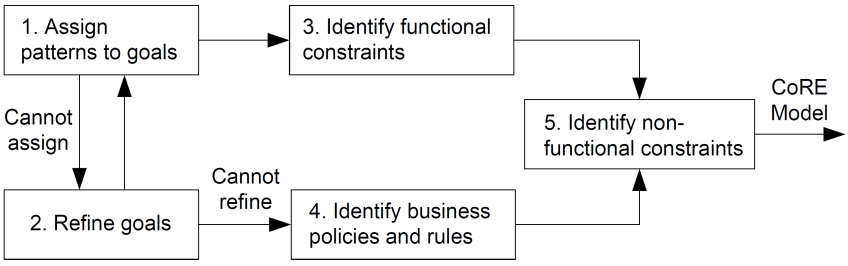


Fig. 2.4. The five steps in CoRE.

Step 1: Assign patterns to goals. CoRE starts by checking if there is a pattern for each goal. A pattern can be assigned to a goal if and only if the operation(s) in the pattern is capable of achieving the goal. There are two ways for this to happen. First, the goal is a direct match to a functionality in the pattern. For example, the goal of searching for a person by name can be directly mapped to the functionality *to find a person by ID or name* in the `PartyManager` archetype [2]. Second, the goal can be refined into subgoals that form a subset of the operations in the pattern. For example, the goal *to manage people information centrally* can be refined into subgoals such as *to add or delete a person*, and *to find a person by ID or name*. These subgoals are a subset of the operations in the `PartyManager` arche-

type. If no patterns can be assigned to the goal, proceed to Step 2 with the goal. Otherwise, proceed to Step 3.

Step 2: Refine goals. This step refines high-level goals into subgoals and repeats Step 1 for each subgoal. To refine a goal, the KAOS goal refinement strategy [8] is used where a goal is refined if achieving a subgoal and possibly other subgoals is among the alternative ways of achieving the goal. For a complete refinement, the subgoals must meet two conditions: (1) they must be distinct and disjoint; and (2) together they must reach the target condition in the parent goal. For example, the goal *to control access to university buildings and resources* is refined into three subgoals: *to maintain up-to-date and accurate person information*, *assign access rights to staff, students, and visitors*, and *verify the identity of a person requesting access*. If these three subgoals are met, then their parent goal is met.

As the refinement aims towards mapping the subgoals to archetypes, the patterns are used to guide the refinement. For example, the leaf goal[3] *to maintain up-to-date and accurate person information* is partly met by the `PartyManager` archetype that manages a collection of people. Hence, the leaf goal is refined into two subgoals: *to manage people information centrally*, and *to automate entries and updates of person information*. A goal cannot be refined if there are no patterns for its subgoals even if it is refined. For example, the goal *to assign access rights to staff, students, and visitors* has no matching patterns as access rights are business specific. In that case, proceed to Step 4.

Step 3: Identify functional constraints. For each pattern that is assigned to a goal, this step identifies functional constraints on the achievement of the goal. This involves asking users of the system about the tasks they depend on the system to carry out, also known as task dependency in i* [35]. These tasks should be significant enough to warrant attention. For example, one of the security guard's task is to compare the cardholders' appearance with their digital photos as they scan their cards. This feature constrains acceptable implementations of the `PartyAuthentication` archetype to those that enable visual checks.

Step 4: Identify business policies and rules. The goals that cannot be further refined are assigned to business policies and rules. This involves searching for policies and rules in the organisation that support the achievement of the goal [21]. For example, the goal *to assign access rights to staff, students, and visitors* is supported by UCL access policies for these user categories. These policies form the basis for access rules that specify the buildings and access times for each of these user categories and their subcategories. For example, undergraduates and postgraduates have different the access rights to university resources.

[3] A leaf goal is a goal without subgoals.

Step 5: Identify non-functional constraints. The final step involves identifying non-functional constraints for all the goals. If a goal is annotated with a non-functional constraint, all its subgoals are also subjected to the same constraint. As such, to avoid annotating a goal and its subgoal with the same constraint, higher-level goals are considered first. For example, the access control system takes people data from other UCL systems, such as the student system and human resource system. As such, for the goal *to maintain up-to-date and accurate person information*, these systems impose data compatibility constraints on the access control system.

The output of the CoRE method is a list of requirements that are separated into the four shearing layers. A visual representation of its output is illustrated in Fig. 2.5. This representation is adopted from the KAOS [8] and i* methods [35]. For example, the goal refinement link means that the three subgoals should together achieve the parent goal, and the means-end link means that the element (functional constraint, non-functional constraint, or pattern) is a means to achieve the goal.

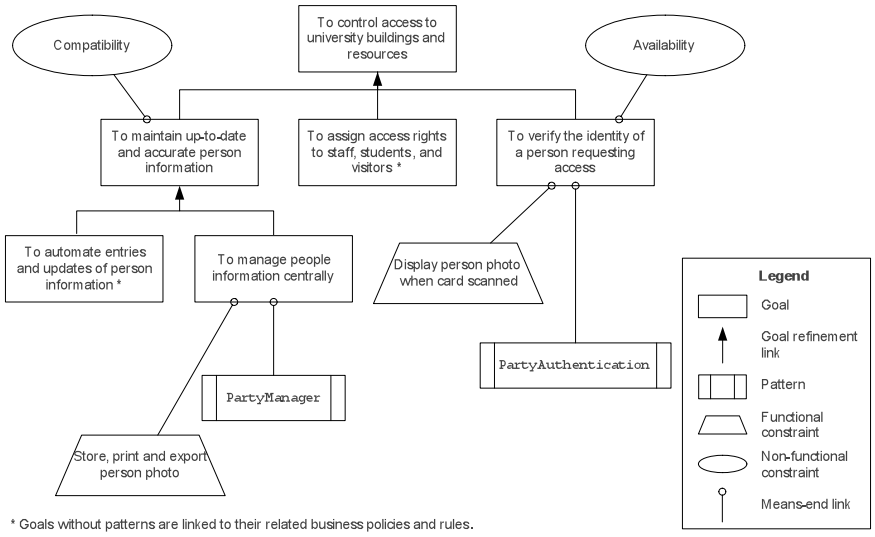


Fig. 2.5. Partial CoRE output for the university access control system.

2.5 Evaluation

CoRE's goal is to separate requirements into layers that change at different rates. The evaluation asks if CoRE can be used to separate requirements into the shearing layers, and if it accurately anticipates the volatility of each shearing layer. The access control system project in University College London is used as a case study to evaluate CoRE. First, CoRE is used to model the initial requirements for the project. Then, the requirements change in the system is recorded over a period of 3.5 years, from the development of the system to the current date after the system is deployed. The result is used to find out the volatility for each layer and when changes in each layer occur in the system lifecycle.

The RALIC Project

RALIC (***R***eplacement ***A***ccess, ***L***ibrary and ***I***D ***C***ard) was the access control system project at University College London (UCL). RALIC was initiated to replace the existing access control systems at UCL, and consolidate the new system with identification, library access and borrowing. RALIC was a combination of development and customisation of an off-the-shelf system. The objectives of RALIC included replacing existing access card readers, printing reliable access cards, managing cardholder information, providing access control, and automating the provision and suspension of access and library borrowing rights.

RALIC was selected as the case study to evaluate CoRE for the following reasons. First, the stakeholders and project documentation were accessible as the system was developed, deployed, and maintained at UCL. Second, RALIC was a well-documented project: the initial requirements and subsequent changes were well-documented. Third, the system development spanned over two years and the system has been deployed for more than two years, providing sufficient time to study change during development as well as maintenance. Finally, RALIC was a large-scale project with many stakeholders [19] in a changing environment, providing sufficient data in terms of requirements and their changes to validate CoRE.

Applying CoRE to RALIC

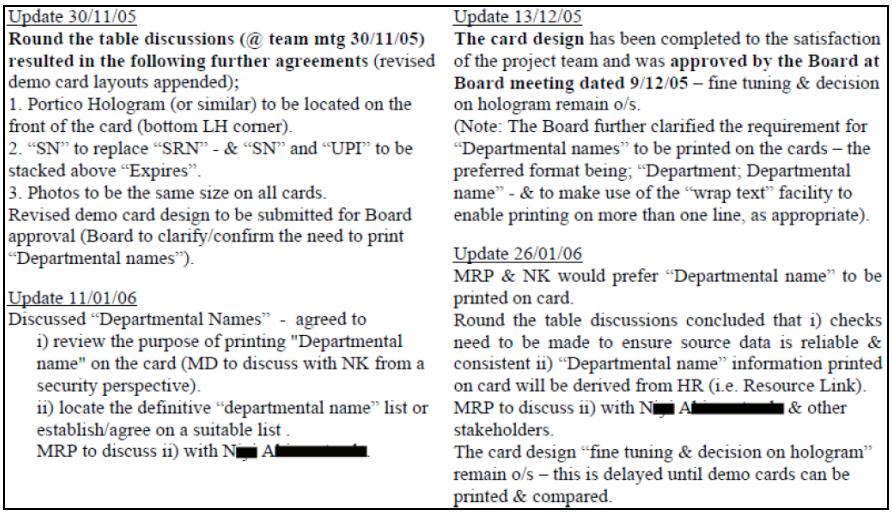
The CoRE method was used to separate the requirements for the RALIC project into the shearing layers. The initial requirements model was built using the requirements documentation signed off by the client as the baseline. The initial requirements model for RALIC consists of 26 elements from the CoRE layers: 3

patterns, 5 functional constraints, 4 non-functional constraints, 5 business policies, and 9 business rules.

To study requirements change, modifications to the requirements documentation after the baseline are considered as a change. There are 3 types of change:

- *Addition*: a requirement is introduced after sign-off.
- *Deletion*: an existing requirement is removed.
- *Modification*: an existing requirement is modified due to changes in stakeholder needs. Corrections, clarifications, and improvements to the documentation are not considered as changes.

As RALIC was an extremely well-documented project, requirements change can be studied retrospectively. During the project, the team members met fortnightly to update their progress and make decisions. All discussions were documented in detail in the meeting minutes as illustrated in Fig. 2.6, by an external project support to increase the objectiveness of the documentation. As such, studying the minutes provided an in-depth understanding of the project, its progress, requirements changes, and their rationale.



Update 30/11/05

Round the table discussions (@ team mtg 30/11/05) resulted in the following further agreements (revised demo card layouts appended);

1. Portico Hologram (or similar) to be located on the front of the card (bottom LH corner).
2. "SN" to replace "SRN" - & "SN" and "UPI" to be stacked above "Expires".
3. Photos to be the same size on all cards.

Revised demo card design to be submitted for Board approval (Board to clarify/confirm the need to print "Departmental names").

Update 11/01/06

Discussed "Departmental Names" - agreed to

i) review the purpose of printing "Departmental name" on the card (MD to discuss with NK from a security perspective).

ii) locate the definitive "departmental name" list or establish/agree on a suitable list .

MRP to discuss ii) with N[redacted].

Update 13/12/05

The card design has been completed to the satisfaction of the project team and was **approved by the Board at Board meeting dated 9/12/05** – fine tuning & decision on hologram remain o/s.

(Note: The Board further clarified the requirement for "Departmental names" to be printed on the cards – the preferred format being; "Department; Departmental name" - & to make use of the "wrap text" facility to enable printing on more than one line, as appropriate).

Update 26/01/06

MRP & NK would prefer "Departmental name" to be printed on card.

Round the table discussions concluded that i) checks need to be made to ensure source data is reliable & consistent ii) "Departmental name" information printed on card will be derived from HR (i.e. Resource Link).

MRP to discuss ii) with N[redacted] & other stakeholders.

The card design "fine tuning & decision on hologram" remain o/s – this is delayed until demo cards can be printed & compared.

Fig. 2.6. An excerpt of RALIC's meeting minutes on card design. Names have been anonymised for reasons of privacy.

RALIC used a semi-formal change management process. During development, minor changes were directly reflected in the documentation. Changes requiring further discussions were raised in team meetings and reported to the project board. Major changes required board approval. Meeting discussion about changes and their outcome (accepted, postponed, or rejected) were documented in the minutes.

During maintenance, team meetings ceased. The maintenance team recorded change requests in a workplan and as action items in a change management tool.

The following procedure was used to record changes. All project documentation related to requirements, such as specifications, workplans, team and board meeting minutes, were studied, as changes may be dispersed in different locations. Care was taken not to consider the same changes more than once. Repeated documentation of the same changes occurred because changes discussed in team meetings can be subsequently reported in board meetings, reflected in functional specification, cascaded into technical specification and finally into workplans. Interviews were also conducted with the project team to understand the project context, clarify uncertainties or ambiguities in the documentation, and verify the findings.

Some statements extracted from the documentation belong to more than one CoRE layer. For example, the statement "for identification and access control using a single combined card" consists of two patterns (i.e., `Person` and `PartyAuthentication`) and a functional constraint (i.e., combined card). In such cases, the statements are split into their respective CoRE layers.

Although the difference between pattern, functional constraint, and non-functional constraint is clear cut, policies and rules can sometimes be difficult to distinguish. This is because high-level policies can be composed of several lower-level policies [21, 3]. For example, the statement "Access Systems and Library staff shall require leaver reports to identify people who will be leaving on a particular day" is a policy rather than a rule, because it describes the purpose of the leaver reports but not how the reports should be generated. Sometimes, a statement can consist of both rules and policies. For example, "HR has changed the staff organisation structure; changes were made from level 60 to 65." Interviews with the stakeholders revealed that UCL has structured the departments for two faculties from a two tier to a three tier hierarchy. This is a UCL policy change, which has affected the specific rule for RALIC, which is to display department titles from level 65 of the hierarchy onwards.

Each change was recorded by the date it was approved, a description, the type of change, and its CoRE layer (or N/A if it does not belong to any layer). Table 2.2 illustrates the change records, where the type of change is abbreviated as A for addition, M for modification, and D for deletion, and the CoRE layers are abbreviated as P for pattern, FC for functional constraint, NFC for non-functional constraint, BP for business policies, BR for business rules, and N/A if it does not belong to any layer. There were a total of 97 changes and all requirements can be exclusively classified into one of the four layers.

Table 2.2. Partial Change Records

Date	Description	Type	Layer
6 Oct 05	The frequency of data import from other systems is one hour (changed from 2 hours).	M	NFC
6 Oct 05	The access rights for students expire three months after their expected course end date (changed from 6 months).	M	BR
18 Oct 05	End date from the Staff and Visitor systems and Student Status from the Student system is used to determine whether a person is an active cardholder.	A	BR
16 Nov 05	Expired cards must be periodically deleted.	A	BP
30 Nov 05	Access card printer should be able to print security logos within the protective coating.	A	FC
8 May 06	The highest level department name at departmental level 60 should be printed on the card.	A	BR
17 Jan 07	The frequency of data import from other systems is 2 minutes (changed from 5 minutes).	M	NFC
2 Apr 07	Replace existing library access control system that uses barcode with the new access control system.	D	BP
1 Jul 08	Programme code and name, route code and name, and faculty name from the Student system is used to determined their access on the basis of courses.	A	BR
15 Aug 08	The highest level department name at departmental level 65 should be printed on the card (changed from 60).	M	BR
1 Jan 09	Introduce access control policies to the Malet Place Engineering Building.	A	BP

Layer Volatility

To evaluate if CoRE accurately anticipates the volatility of each shearing layer, the change records for RALIC (Table 2.2) is used to calculate each layer's volatility. The volatility of a layer is the total number of requirements changes divided by the initial number of requirements in the layer. The volatility ratio formula from Stark *et al*. [31] is used (Eq. 2.1).

$$volatility = \frac{Added + Deleted + Modif\ ied}{Total}, \qquad (2.1)$$

where *Added* is the number of added requirements, *Deleted* is the number of deleted requirements, *Modified* is the number of modified requirements, and Total is the total number of initial requirements for the system. Volatility is greater than 1 when there are more changes than initial requirements.

Using Eq. 2.1 to calculate the volatility for each layer enables the comparison of their relative volatility. As expected, patterns are the most stable, with no changes over 3.5 years. This is followed by functional constraints with a volatility ratio of 0.6, non-functional constraints with a volatility ratio of 2.0, and business policies and rules with a volatility ratio of 6.4. The volatility ratio between each layer is also significantly different, showing a clear boundary between the layers. Business policies and business rules have similar volatility when considered separately: policies have a volatility ratio of 6.40 and rules 6.44.

Timing of Change

The volatility ratio indicates the overall volatility of a layer. To understand *when* the changes occur, the number of quarterly changes for each layer is plotted over the duration of the requirements change study, as illustrated in Fig. 2.7.

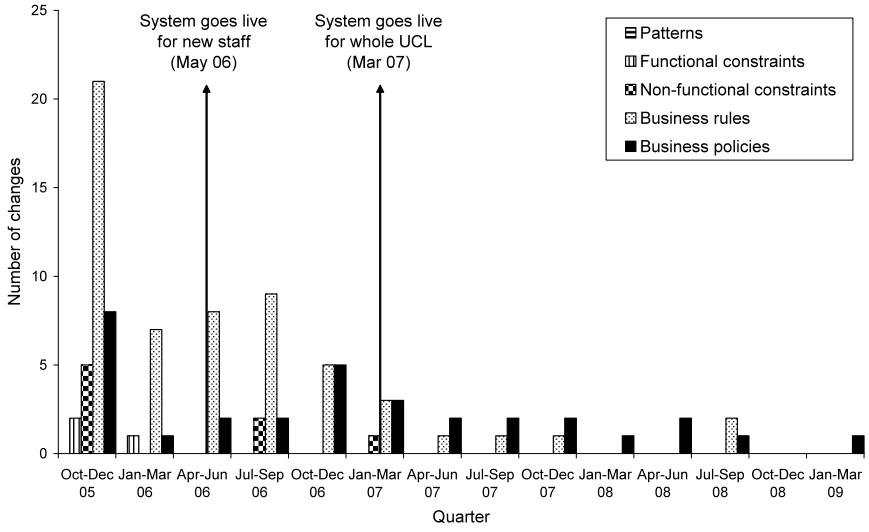


Fig. 2.7. Quarterly requirements changes for each layer.

The quarter Oct-Dec 05 has the highest number of changes for functional constraints, non-functional constraints, business policies and rules because the requirements elicitation and documentation were still in progress. The project board had signed off the high-level requirements, but the details surrounding access rights and card processing were still under progress. Many of the changes were

due to better understanding of the project and to improve the completeness of the requirements.

Consistent with the existing literature (e.g., [7]), missing requirements surfaced from change requests after the system was deployed. The system went live first for new staff in May 06 and then for the whole of UCL in March 07. Each time it went live, the number of requirements change increased in the following quarters.

A rise in policy change in quarters Oct-Dec 05 and Jan-Mar 07 was followed by a rise in rule change in the following quarters, because business rules are based on business policies. As more than one rule can be based on the same policy, the number of changes in rules is naturally higher than that of policies. Nevertheless, after the system went live, policy changes did not always cascade into rule changes. For example, application of the access control policy to new buildings required only the reapplication of existing rules.

Interestingly, the quarterly changes for business rules resemble an inverse exponential function, as the number of changes was initially large but rapidly decreased. In contrast, the quarterly changes for business policies shows signs of continuous change into the future. Rules suffered from a high number of changes to start with, as the various UCL divisions were still formulating and modifying the rules for the new access control system. After the system went live, the changes reduced to one per quarter for three quarters, and none thereafter. One exception is in quarter Jul-Sep 08, where UCL faculty restructuring had caused the business processes to change, which affected the rules. Nevertheless, these changes were due to the environment of the system rather than missing requirements.

Implications

CoRE produces requirements models that are adequate without unnecessary details because leaf goals are either mapped to archetypes, which are the essence of the system, or to business policies and rules, ensuring that business specific requirements are supported by business reasons. CoRE does not rely on domain experts because the archetypes capture requirements that are pervasive in the domain. The requirements models are complete and pertinent because all the requirements in RALIC can be classified into the four CoRE layers. Also, RALIC stakeholders could readily provide feedback on CoRE models (e.g., Fig. 2.5), showing that the model is easy to understand.

As CoRE is based on goal modelling, it inherits their multi-level, open and evolvable, and traceable features. CoRE captures the system at different levels of abstraction and precision to enable stepwise elaboration and validation. The AND/OR refinements enables the documentation and consideration of alternative options. As CoRE separates requirements based on their relative volatility, most changes occur in business policies and rules. The rationale of a requirement is

traceable by traversing up the goal tree. The source of a requirement can be traced to the stakeholder who defined the goal leading to the requirement.

Finally, CoRE externalises volatile business policies and rules. As such, the system can be designed such that software architecture components that implement these volatile requirements are loosely coupled with the rest of the system. For example, in service-oriented architecture, these components can be implemented such that changes in business policies are reflected in the system as configuration changes, and changes in business rules are reflected as changes in service composition [18]. This minimises the disruptive effect of changing requirements on the architecture.

2.6 Future Work

The study is based on a single project, hence there must be some caution in generalising the results to other projects, organisations, and domains. Also, the study assumed that all requirements and all changes are documented. Future work should evaluate CoRE on projects from different domains, and in a forward looking manner, i.e., anticipate the change and see if it happens. As RALIC is a business system, enterprise archetype patterns were used. Future work should investigate the use of software patterns in other domains, such as manufacturing or medical domains. Finally, future work should also investigate the extent of CoRE's support for requirements engineers who are less experienced.

The requirements changes that CoRE anticipates are limited to those caused by the business environment and stakeholder needs. But requirements changes can be influenced by other factors. For example, some requirements may be more volatile than others because they cost less to change. In addition, CoRE does not consider changes due to corrections, improvements, adaptations or uncertainties. Future work should consider a richer model that accounts for these possibilities, as well as provide guidance for managing volatile requirements.

CoRE anticipates change at the level of a shearing layer. But among the elements in the same layer, it is unclear which is more volatile. For example, using CoRE, business rules are more volatile than functional constraints, but it is unclear which rules are more likely to change. Future work should explore a predictive model that can anticipate individual requirements change and the timing of the change. This could be done by learning from various attributes for each requirement such as the number of discussions about the requirement, the stakeholders involved in the discussion and their influence in the project, and the importance of the requirement to the stakeholders. Much of these data for RALIC have been gathered in previous work [17].

2.7 Conclusion

This chapter has described CoRE, a novel expert independent method that classifies requirements into layers that change at different rates. The empirical results show that CoRE accurately anticipates the volatility of each layer. From the most stable to the most volatile, the layers are patterns, functional constraints, non-functional constraints, and business policies and rules.

CoRE is a simple but reliable method to anticipate change. CoRE has been used in the Software Database project[4] to build a UCL wide software inventory system. Feedback from the project team revealed that CoRE helped the team better structure their requirements, and gave them an insight of requirements that were likely to change. As a result, their design and implementation prepared for future changes, thus minimising the disruptive effect of changing requirements to their architecture.

Acknowledgments The authors would like to thank members of the Estates and Facilities Division and Information Services Division at University College London for the RALIC project documentation, their discussions and feedback on the requirements models, as well as Peter Bentley, Fuyuki Ishikawa, Emmanuel Letier, and Eric Platon for their feedback on the work.

References

[1] Anton A. I. (1996) Goal-based requirements analysis. In Proceedings of the 2nd International Conference on Requirements Engineering, pages 136-144.
[2] Arlow J. and Neustadt I. (2003) Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. Addison-Wesley Professional.
[3] Berenbach B., Paulish D. J., Kazmeier J., Daniel P., and Rudorfer A. (2009) Software Systems Requirements Engineering: In Practice. McGraw-Hill Osborne Media.
[4] Boehm B. W. (1981) Software Engineering Economics. Prentice Hall.
[5] Brand S. (1995) How Buildings Learn: What Happens After They're Built. Penguin Books.
[6] Chung L., Nixon B. A., Yu E., and Mylopoulos J. (1999) Non-functional Requirements in Software Engineering. Springer.
[7] Cockburn A. (2002) Writing Effective Use Cases. Addison-Wesley Professional.
[8] Dardenne A., van Lamsweerde A., and Fickas S. (1993) Goal-directed requirements acquisition. Science of Computer Programming, 20(1-2):3-50.
[9] Foote B. and Yoder J. (2000) Big ball of mud. Pattern Languages of Program Design, 4(99):654-692.
[10] Harker S. D. P., Eason K. D., and Dobson J. E. (1993) The change and evolution of requirements as a challenge to the practice of software engineering. In Proceedings of the IEEE International Symposium on Requirements Engineering, pages 266-272.
[11] IEEE Computer Society. (2000) IEEE Recommended Practice for Software Requirements Specifications.
[12] International Organization for Standardization. (2001) Software Engineering - Product Quality, ISO/IEC TR 9126(1-4).

[4] http://www.ucl.ac.uk/isd/community/projects/azlist-projects

[13] Jacobson I. (1995) The use-case construct in object-oriented software engineering. Scenario-based Design: Envisioning Work and Technology in System Development, pages 309-336.
[14] Jones C. (1996) Strategies for managing requirements creep. Computer, 29(6):92-94.
[15] Kotonya G. and Sommerville I. (1998) Requirements Engineering. Wiley.
[16] Leffingwell D. (1997) Calculating the return on investment from more effective requirements management. American Programmer, 10(4):1316.
[17] Lim S. L. (2010) Social Networks and Collaborative Filtering for Large-Scale Requirements Elicitation. PhD Thesis, University of New South Wales. Available at: http://www.cs.ucl.ac.uk/staff/S.Lim/phd/thesis_soolinglim.pdf.
[18] Lim S. L., Ishikawa F., Platon E., and Cox K. (2008) Towards agile service-oriented business systems: A directive-oriented pattern analysis approach. In Proceedings of the 2008 IEEE International Conference on Services Computing, Vol. 2, pages 231-238.
[19] Lim S. L., Quercia D., and Finkelstein A. (2010) StakeNet: Using social networks to analyse the stakeholders of large-scale software projects. In Proceedings of the 32nd International Conference on Software Engineering, Vol. 1, pages 295-304.
[20] Mens T. and Galal G. (2002) 4th Workshop on Object-oriented Architectural Evolution. pages 150-164. Springer.
[21] Object Management Group. (2006) Business Motivation Model (BMM) Specification, Technical Report dtc/060803.
[22] O'Neill R. V., DeAngelis D. L., Waide J. B., and Allen T. F. H. (1986) A Hierarchical Concept of Ecosystems. Princeton University Press.
[23] Papantoniou B., Nathanael D., and Marmaras N. (2003) Moving Target: Designing for Evolving Practice. In HCI International 2003.
[24] Robertson S. and Robertson J. (2006) Mastering the Requirements Process. Addison-Wesley Professional.
[25] Ross R. G. (2003) Principles of the Business Rule Approach. Addison-Wesley Professional.
[26] Salthe S. N. (1993) Development and Evolution: Complexity and Change in Biology. MIT Press.
[27] Simmonds I. and Ing D. (2000) A shearing layers approach to information systems development, IBM Research Report RC21694. Technical report, IBM.
[28] Sommerville I. (2004) Software Engineering. Addison-Wesley, 7th edition.
[29] Sommerville I. and Sawyer P. (1997) Requirements Engineering: A Good Practice Guide. John Wiley & Sons.
[30] Standish Group. (1994) The CHAOS Report.
[31] Stark G. E., Oman P., Skillicorn A., and Ameele A. (1999) An examination of the effects of requirements changes on software maintenance releases. Journal of Software Maintenance: Research and Practice, 11(5):293-309.
[32] van Lamsweerde A. (2001) Goal-oriented requirements engineering: A guided tour. In Proceedings of the 5th IEEE International Symposium on Requirements Engineering, pages 249-262.
[33] van Lamsweerde A. (2009) Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley & Sons.
[34] Wiegers K. (2003) Software Requirements. Microsoft Press, 2nd edition.
[35] Yu E. S. K. (1997) Towards modelling and reasoning support for early-phase requirements engineering. In Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, pages 226-235.