Using a Rule-Base Approach and Responsibility Modeling for Automatic Software Evolution

by

Ming-Jen Huang

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2008

Abstract

The purpose of this research work is to improve software evolution by managing the complex relationships between abstractions of different development stages. To this end, we propose and implement an automation approach for managing these relationships. This approach is based on the idea of capturing and reusing various types of relationships between abstractions.

A program realizes different types of high-level abstractions. As more functions are added to the program, the realization relationship between the program and the high-level abstractions conceived in the development process becomes more complex. To evolve a program without degrading its quality, managing this complexity is the key point. To this end, in this research work we propose a new development approach, which is based on three theories. (1) First, to eliminate the gap between different worlds in software development process, we use a single-type paradigm for modeling abstractions that are created in different worlds but are also related at the same time. (2) Second, to simplify the evolution of the relationships among abstractions, we propose directly creating a program by reusing previously considered development knowledge of relationships among abstractions. More specifically, a program is constructed from the modules of the relationships of abstractions which are conceived in the development process and are recorded by the single-type paradigm in (1). (3) Third, we propose using rule engine for implementing a tool for automating software evolution by reusing and composing the modules mentioned in (2). The automation provided by this approach is for the following three evolution scenarios: (a) when the given business processes are evolving, (2) when the realization-development knowledge is evolving, and (3) when a different implementation technology is adopted. To evaluate the effectiveness of the proposing approach, a case study with three software systems is conducted.

In this dissertation, we describe the construction of the proposing approach. In the first step, the basic framework is constructed. This framework helps developers to capture development knowledge they acquire in the development process. It includes a modeling language and a set of graphical notations. We then describe how the modules of relationships among abstractions can be used to construct/evolve a program. In the second step, the

implementation for automated program construction/evolution is developed. This implementation provides the features of development knowledge modeling and program construction/evolution automation. Finally, a case study is conducted. The results of the case study provide the support for the proposing three theories for software evolution.

The evaluation results show that a single-type paradigm by using responsibility can be effectively used to describe the relationships of abstractions within the four worlds. The modularization of development knowledge can effectively capture how developers design realization of abstractions of different worlds. Finally, a rule engine encodes the development knowledge for inferring the development of system responsibilities, object responsibilities, and program responsibilities.

Acknowledgments

This research work is supported by many people. I would like to thank to Professor Takuya Katayama for his kindly guidance, encouragement, and support in many different forms. I am very lucky to have him to be the supervisor of my master and PhD study.

I would like to thank to all committee members: Professor Motoshi Saeki, Professor Koichiro Ochimizu, Professor Tomoji Kishi, Associate Professor Masato Suzuki, and Associate Professor Toshiaki Aoki. Your comments are invaluable to me.

I would like to thank to my colleagues, especially Assistant Professor Kenrou Yatake, Dr. Rami Yared, Dr. Samia Souissi, Mr. Nǎixué Xióng, and Ms Yàn Yáng.

I would like to thank to Professor Leon J. Osterweil at University of Massachusetts Amherst and every people I met while I was there. This was the most exciting moment I had so far.

I would like to thank to my family in Taiwan. Especially, I would like to thank to Mom for making me love to read and to learn from my childhood. I would also like to thank to my mother-in-law for her kindly help while I were finishing this dissertation. I would like to thank Mr. and Mrs. Hamada in Nagoya for treating me like your family member.

Last but not least, I would like to thank to my wife Lisa, our lovely son Wei-Wei, and our coming baby. You keep me moving forward.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Problem

1.1.1 Background

A business software system is usually developed as a staged-process. Among other activities, in each stage developers conceived abstractions to realize abstractions created in a previous stage. For example, for developing a business system, developers firstly define business tasks and business actors in a business process. From here, user requirements are defined to realize these business activities, software design is created to realize user requirements, and program is pondered to realize software design. Finally, a program that realizes all these high-level abstractions is implemented. In this process, there are many relationships designed by developers. We can see realization of abstractions between two stages, collaboration of entities within a stage, or a constraint on the realization or collaboration. As customers request more functions, a program is more bounded to abstractions conceived in the process. It becomes harder to manage these relationships to evolve software. The overall result is a quality-degraded program [1].

Previous research work focuses on different aspects of this problem. One concept that is considered an effective approach for preventing a quality-degraded program is reusing. This concept is closely related to modularization. In software engineering, we have various modularization paradigms for creating implementation-based artifacts, such as functions in function oriented programming, and objects in object-oriented programming (OOP). These paradigms eliminate repetition when creating a program and help developers focus on a small area of development without being bothered by other unrelated issues. The construction repetition can be minimized because a function or a class (a class is the definition of an object) can be reused many times to realize high-level abstractions. Another similar concept is component-based reusing, such as COM on Windows [2] or EJB on Java [3]. Different from the reusing paradigms introduced so far, which are at source-level and

for single-platform reusing, the component-based reusing is binary-level, single/multi-platform reusing.

Even the repetition of implementation can be reduced by the above approach, but one kind of repetition that is rarely been considered is the abstraction realization between different development stages. This kind of repletion can be observed in a development project. It can be easily observed that some similar implementation modules are always created for realizing some similar high-level abstractions. More specifically, certain functions, objects, or a fragment of code are reused collectively and repeatedly for realizing some similar high-level abstractions. Developers possibly only customize an existing solution to realize high-level abstractions rather than creating a new solution every time. For example, some similar business tasks are always realized by using similar object design, and constructed in similar ways. However, current development methodologies or programming paradigms do not provide formal support for reusing these customizable solutions. Productivity provided by such a support is overlooked. The reusing mechanism only focuses on the expected behavior provided by the programming modules, rather than the high-level purpose of the modules construction. In the current ever-changing business environment, design knowledge, and implementation technology, the management of these relationships become more complex and more important.

In the following sections, the problem and the gap that motivates this research work are discussed in details. The solution we propose in this research work is also introduced.

1.1.2 Problems and Gap

There are two problems when overlooking abstraction relationships. First, without such information, developers are hard to answer such a simple question: “could you please tell me which part in a program implements this requirement?” It is also hard to guarantee that high-level and low-level abstraction is consistently constructed when evolution happens. Second, we have to reinvent (or forget) a good design of abstraction relationships to solve some similar problems.

Although the abstraction relationship plays such an important role in software development, the truth is that current technologies and practices usually focus on proposing better approaches from a diagonal direction. That is, developers can easily add or modify a method or an object to a program; however they lack an explicit support for relating this newly added or modified part with other part in the program. Besides, they lack the support of comprehending the relationships between the change of implementation-level artifacts and high-level abstraction, such as requirements and software design.

A different strategy that fills this gap should be proposed. This approach should value the importance of the relationships among abstractions. By this strategy, developers can model the relationships conceived for different systems. By this model, developers can focus on designing a small area of these relationships each time. Each small area of this model is encapsulated as an independent unit. A solution for any given problem in the development process can be created by combining these reusable units. The construction of a program is simply the assembly of the units. This unit does not only provide as a reusable knowledge for constructing a program, but also the information for comprehending the design of a program.

Therefore, in this research work, a development approach that focuses on the modularization of abstractions is proposed. In this dissertation, we describe how this idea forms an approach for software evolution and how the implementation of this approach is applied to the automation of program construction/evolution.

1.2 Overview of the solution

The problems and the gap motivate us to propose a new approach for software evolution. This approach, as we mentioned before, values the importance of the abstraction relationships. By this approach, a program is constructed and evolved by the application and combination of reusable knowledge. Eventually, an implementation of this approach is created for automating program construction/evolution.

1.2.1 Basic idea of the solution

A program can be considered as a big solution to a big problem of the real world. A relationship between high-level and low-level abstractions of this program can be considered as a pair of one small problem and one small solution, which may belong to two different development stages. We use the term, worlds, to represent these stages. It is because a stage usually has abstractions that are specified to that stage which form as a world. The problem of high-level abstractions in one world is solved, or realized, by the solution of low-level abstractions in another world. Practically, there are patterns when defining the realization between any two worlds. Developers reuse or customize exiting relationship to create a solution to solve similar problems. This phenomenon is especially true to the business domain. In business domain, we can observe

- Highly repetitive business processes. There are many similar business tasks in different business processes.
- Structural system design. Developers usually use construct a system in a similar way, for example, the application of three-layered architecture: presentation layer, business-logic layer, and integration layer.
- Abundant object-design solution. There are many reusable solutions been considered for the object-design problems.

Based on these observations, we make an assumption that in the business domain it is possible to derive a pattern from a collection of similar abstraction relationships. This pattern becomes an effective mechanism for creating other concrete relationship when constructing/evolving a program. This proposing approach is constructed on this assumption. By considering a program as a solution for the problem of business processes, this program can be constructed/evolved by consulting these patterns. Moreover, when the problems, solutions, and the realization relationships connecting them are encoded into a computable form, a program can be automatically constructed or evolved.

Each pattern, which is called a parameterized realization unit (PRU) in this approach, is a reusable asset for constructing/evolving a program. PRUs are used to stored humans' knowledge about abstraction realization. A PRU represents a piece of abstraction realization-development knowledge between two worlds. It also contains other relationships, i.e. collaboration and constraints, that are related to this realization. It works as a template which can be instantiated for creating a concrete realization relationship, where each instantiated instance provides a "small" solution to a "small" problem. From this instance, developers know what abstractions (solution) should be created in one world, when they encounter some abstractions (problem) in another world. The collection of these instances relates all abstractions conceived in the development process of a program. The evolution of a program becomes the addition, removal, or replacement of the instances of these patterns. When developers learn more about the business domain, they can construct/evolve a program more productive by only reusing the PRUs.

The problem and the solution pair effectively encode two types of information. The first type is the condition of a solution. That is, when one solution that represents what abstractions should be created when one problem is encountered. Therefore, the first type of information tells developers when they should reuse a unit. The second type of information tells developers what abstractions they should create when this unit is reused.

1.2.2 Fundamental theories of the solution

Before proposing this approach and implementing it as a tool for the evolution automation, there are some fundamental issues that should be solved.

(1) First, the current multi-paradigm practice for software development may hamper the creation of a PRU. Currently, different types of paradigms or concepts are used for abstraction description. For example, while developers use a process oriented language to describe a business process, they use different concepts, such as software objects for realizing this business process. (2) Second, a pattern, i.e. a PRU helps human developers record and reuse their development knowledge about abstraction realization for constructing/evolving a program. This unit also helps machines for the same purpose with a

step further beyond the manual way by humans. By encoding PRUs, and the problems and solutions involving in the units, a machine knows how to construct/evolve a program automatically. Therefore, (3) the third issue is to find an efficient platform for implementing our idea. We need a platform that helps developers directly encode their development knowledge in a computable form. This platform should also help us create the actions for reusing the realization development knowledge.

The answers to the three issues are three fundamental theories, which are illustrated in Figure 1-1. (1) First, to eliminate the gap between different worlds, we propose responsibility modeling, a modeling approach based on single-type paradigm, responsibilities. Responsibility in our approach is not only used for designing what work should be done by software object [4][5], but also for modeling the tasks that should be performed by entities of different worlds and different types of information that should be processed by the entities. Most importantly, responsibilities provide a good abstraction for describing the relationships among different entities. (2) Second, to simplify to manage the abstraction relationships of a program, we propose capturing the connections among abstraction as reusable and composable knowledge by the paradigm of responsibility. By this theory, these connections become the first-class citizen for constructing and evolving a program. (3) Third, to automate the reusing and composing of knowledge, we propose using rule-based engine by encoding the realization-development knowledge. A tool implementing on rule-based engine can automatically infer a program as the solution to the problem of the given business processes.

The combination of the three proposing theories is a development approach, called ***Responsibility-Steering Development*** (RSD for short), will fill the gap we mentioned in Section 1.1.2. The implementation of this approach is tool, called ***RSDTools***, for automating program construction/evolution.

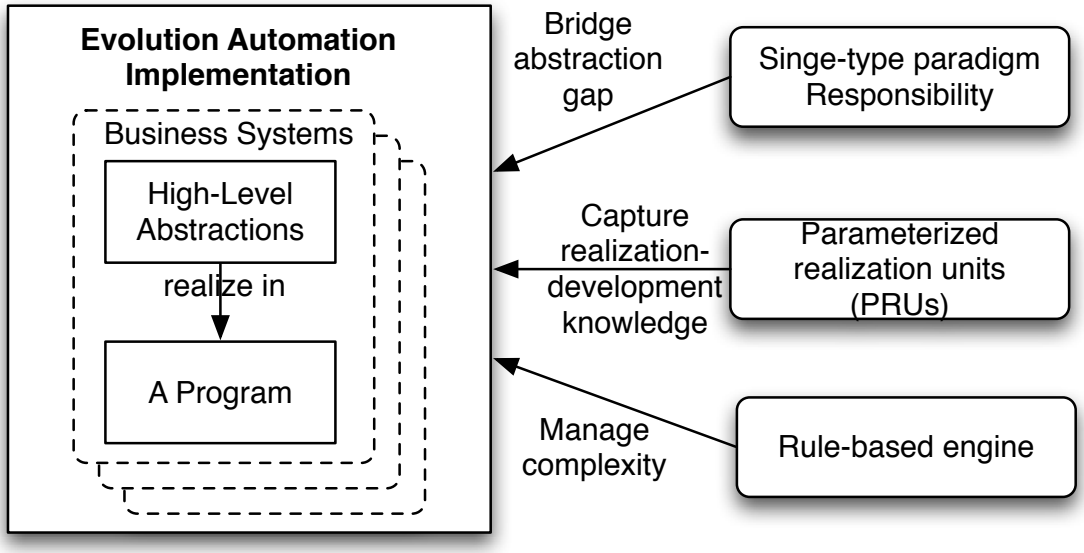


Figure 1-1. Three fundamental theories for Evolution Automation

1.2.3 Scope of the solution

The scope of our solution can be discussed from two aspects:

First, its application domain is limited to business domain. The proposing approach focuses on the modeling of the realization of humans' responsibilities in a business by a program. The automation support only applies to the evolution that happens between humans' responsibilities in a business and the program that automates the performing of these responsibilities.

Second, by using RSD, the development of a business system can be *dynamically satisfied* by using the collection of realization-development knowledge. Dynamically satisfaction of software development by using realization-development knowledge represents that developers can freely add new development knowledge to realize any unrealized business responsibility without invalidating current realization.

This capability is limited to the following three evolution scenarios: business-processes evolution, realization-development knowledge evolution, and technology evolution.

- **Business-processes evolution:** Assuming there is a program, which has been constructed for realizing some business responsibilities by using a collection of development knowledge, this program can be automatically evolved when the given business responsibilities are added, modified, or removed.
- **Realization-development knowledge evolution:** Assuming there is a program, which has been constructed for realizing some business responsibilities by using a collection of development knowledge, this program can be automatically evolved when the collection of development knowledge is added, modified, or removed.
- **Technology evolution:** Assuming there is a program, which has been constructed for realizing some business responsibilities by using a collection of development knowledge; this program is automatically evolved when the underlying implementation technology is changed.

1.2.4 Construction of the solution

We construct RSD in three steps (see Figure 1-2).

First, the basic framework of RSD is designed. This framework is used to help developers to capture realization-development knowledge they acquire in the development process. It includes a modeling language for capturing the realization-development knowledge. It also includes the definition of four connected worlds, where each world corresponds to one stage and provides a distinct context for creating abstractions. Finally, it defines the process of using many small pieces of realization-development knowledge for constructing/evolving a program.

Second, a tool for supporting the automated construction/evolution of a program is developed. This tool is constructed based on the idea and the fundamental theories mentioned before. Developers can use it to capture realization-development knowledge and to model business processes by using the modeling language designed in the first step. It

automates the construction/evolution of a program under the three scenarios described in Section 1.2.3.

Finally, a case study which includes three business systems is developed. We use this case study to evaluate the effectiveness of the proposing approach. More details about these three systems are given in Section 1.2.5.

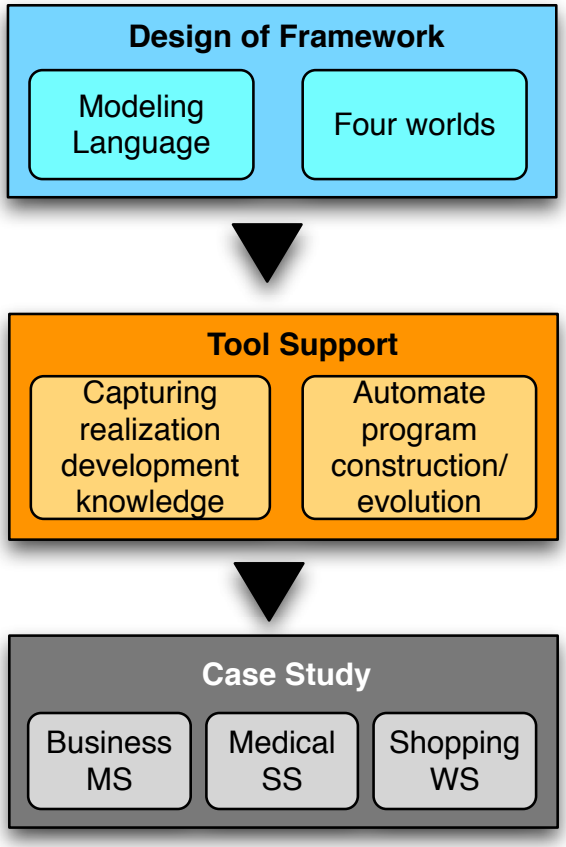


Figure 1-2. Conceptual structure of the construction process

1.2.5 Case Study of the solution

A case study for evaluating the effectiveness of the proposing approach is conducted. This case study includes the development of three software systems, a business-process management system (called Business-MS), a medical supporting system (called Medical-

SS), and shopping-mall-on-web system (called Shopping-WS). The first system has been commercially deployed. The second system is a research based on the paper [6]. The third system will be commercially deployed in future. These three systems verify the claim that made in Section 1.2.3. This claim is that in the three evolution scenarios, the development of a business software system can be dynamically satisfied by the collection of realization-development knowledge. Dynamically satisfaction of software development by using realization-development knowledge represents that developers can freely add new development knowledge to realize any unrealized business responsibility without invalidating current realization under the three evolution scenarios mentioned above. Therefore, this case study is intended to verify the business-process evolution and realization-development knowledge. The technology evolution is verified by using development technologies to create different variations from the same set of business-process responsibilities.

To simulate different evolution scenarios, the same set of requirements of the three systems are implemented by using different technologies. The first (Business-MS) and the second (Medical-SS) systems have two variations. One is implemented by using JavaServer Pages (JSP) [7] and JavaBeans [8][9]. JSP is for information visualization and JavaBeans is for information processing logic. The other is implemented by using JBoss Seam [10][11], which is a new programming model for creating Java enterprise system. However, the third system (Shopping-WS) is only implemented by using JBoss Seam.

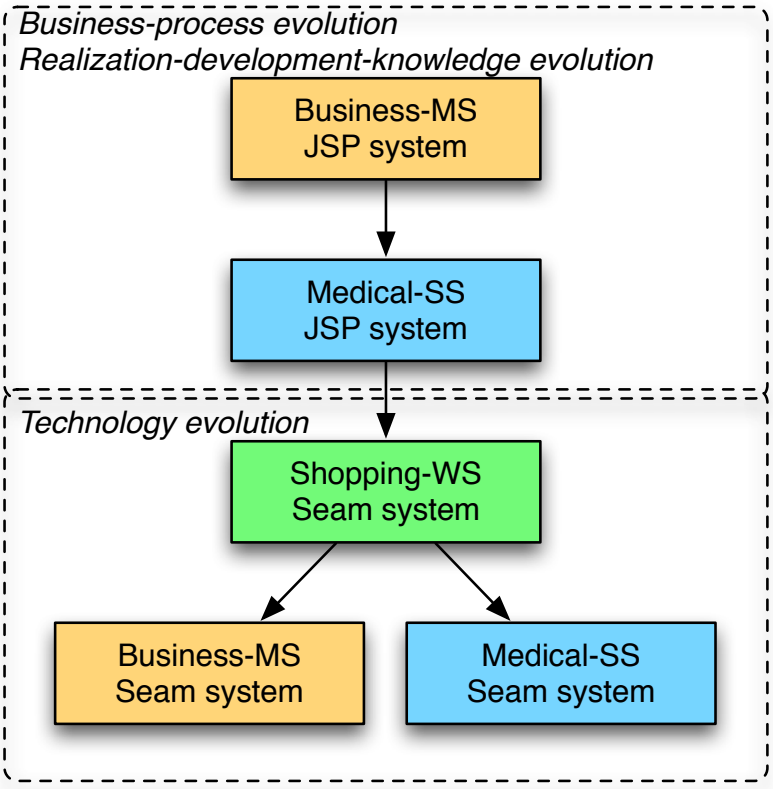


Figure 1-3. Creation order of the three systems

The creation order of these three systems implies the evolution process, which is shown in Figure 1-3. The creations of the first variation of Business-MS and Medical-SS illustrate business-process evolution and realization-development-knowledge evolution. The creations of Shopping-WS and the second variation of Business-MS and Medical-SS illustrate technology evolution. We use the PRUs created for the third system to develop the second variation of Business-MS and Medical-SS. Table 1-1 summaries the implementation technologies used by each system.

Table 1-1. Implementation technology of the evaluating systems.

Evaluating System	Implementation Technology	
	JSP	JBoss Seam
Business-MS		
1st variation	✓	
2nd variation		✓
Medical-SS		
1st variation	✓	
2nd variation		✓
Shopping-WS		✓

1.3 Organization of the dissertation

This paper is organized as follows:

In Chapter 1, we describe research topic. We state the problem and gap that motivates us to propose the approach. We describe the proposing approach for software evolution by explaining its basic idea and fundamental theories. We also state the intended evolution scenarios of the proposing approach. Finally, the construction and the evaluation of this approach are summarized.

In Chapter 2, we review some previous work that is related to our research topic.

In Chapter 3, we describe the basic framework that helps developers to capture realization-development knowledge they acquire in the development process. It includes a modeling language for capturing realization-development knowledge. A supplementary set of graphical notations is also provided for visualizing the captured development knowledge.

In Chapter 4, we describe how program construction/evolution is achieved by reusing the realization-development knowledge. The reuse is centered about an idea called parameterized realization unit (PRU), which is a customizable realization relationship.

PRUs are used to store humans' knowledge for reusing. We show that how a program constructed by using PRUs is also capable of to be evolved by the same mechanism of PRUs.

In Chapter 5, we describe the implementation of the supporting tool. We show the features the tool provides, the structure the tool is constructed, and the internal work it performs for the automation of program construction/evolution. This tool does not only provide the modeling of realization development-knowledge, but also has a rule-based engine integrated for program construction/evolution automation. We show how rules are implemented for this automation.

In Chapter 6, we describe the evaluation of the proposing approach. We show the statistics of the evaluating results, which characterizes the novelty of this approach.

In Chapter 7, we summarize this dissertation and future work.

Chapter 2 Related Work

Software evolution becomes an emerging area of research work. Lehman and Ramil [12] discussed the definition of software evolution. From their definition, we can separate the study of software evolution into the *means* and the *observation*. The former concerns *how* and the later concerns *when* and *what*. There are fewer research work about the later [13, 14, 15, 16, 17, 18, 19, 20, 21]. The topics of the former are various. We have program evolution by refactoring [22, 23, 24] for source code evolution. Another important area is higher-level abstraction evolution, which concerns more on requirements or design aspect of a program [25, 26, 27, 28]. There is also some research work about external environment (e.g. business, work, etc.) evolution [29. 30].

Our research work limits its applicability the evolution of higher-level abstractions and to the business domain. One important characteristic of the software systems of this domain is that they concern the real-world business activities. As suggested by Lehman and Fernandez-Ramil [31], the systems for the business domain, which were also called E-type systems by their work, have an important characteristic that their behavior must satisfy the operational context. That is, they must exhibit the behavior defined in terms of computer abstraction that satisfies user requirements defined in terms of the real-world abstraction. Synchronizing the two worlds reveals one of the challenges in the study of software evolution [32].

This is such a complex issue that is approached by previous work from different aspects. Since it is not possible to review all of the aspects in this dissertation, we limit the review of previous work to those that are possible to solve the problem of dis-synchronized between real-world and computer-world abstractions. The following research aspects are discussed.

2.1 Model-driven development

The first is an attempt that tries to use a set of universal rules for mapping between the real-world and the computer world. This set of universal rules will transform any given real-world problem to any computer-world solution. This approach is usually called model-driven development (MDD), since the problem domain, the solution domain, and the mapping rules are defined under a metamodel [33][34][35]. A metamodel is a model for defining other models, which can be used to define the concepts for describing the facts of the real-world (i.e. the problem domain), the computer-world (i.e. the solution domain), and the mapping between these two worlds. Since both worlds are defined in terms of the same modeling paradigm, the mapping rules can be easily created. One example that has been discussed frequently is MDA (model-driven architecture) [36].

We use MDA to discuss the general approach adopted by MDD. The single most important element of MDD is the transformation definition between models. In MDA, there is a standardized metamodel called MOF for defining transformation. Consequently, the source and the target of transformation is also defined in this meteamodel. The significance of MDA transformation is that abstractions of the real-world problem should be separated by the computer-world. That is, the modeling of the real-world problem has no concerns of the solution of the problem. The solution is derived i by the transformation definition. It is a very important characteristic because such the separation cannot be easily achieved by other traditional approach. The metamodel MOF define the scope of modeling different worlds. Under MDA's terminology, the source of transformation is called PIM (platform-independent model), and the target of transformation is called PSM (platform-specific model). A PIM is always impendent from some form of abstractions. For example, a model for describing business processes is independent from how a system is automated by a software system. Therefore, a model that describes business processes is a PIM and the model that describes the automation of business processes by a software system is a PSM. The meaning of this approach to the synchronization of the two worlds under the context of software evolution is apparently. Since there is a transformation definition that is

universally capable to transform between any two models which belong to one pair of two specific domains, the solution for a continually evolving problem domain can always be inferred.

Since the implementation of model transformation, i.e. the actual logic that use the transformation definition to create the target model from the source model, is not specified by MDA, research work and industrial products based on MDA's standards or concepts are abundant, and usually has different focuses. At the same time, it is also hard to clearly distinguish an approach that adheres to MDA speciation and those that are merely based on the concept of model-driven development. As Sendall and Kozaczynski [37] summarize the various mechanisms into there are three types, which are direct model manipulation, intermediate representation, and transformation language support. Examples of the first type are some commercial tools such as Rational XDE, which uses a set of VB API for model manipulation. Action language [38] also falls into this category. Examples of the second type include XML-based representations such as XMI [39].

Some work are reviewed below. Since this dissertation is not on the topic of MDA but about software evolution in general, the work reviewed below are not limited to *MDA-compliant*.

Arlow et. al. [40] describes a transformation approach called archetype patterns that each archetype pattern specifies a mapping rule between the problem (analysis, design) and the solution (design) domains. An archetype pattern may have various variations for fitting in different context. They describe their approach can be defined in MDA's standards and automated by tools. Wegmann et. al. [41] proposes combining three elements, MDA, enterprise architecture (EA), and the living system theory (LST) to integrate different models in a hierarchical structure that includes business, organization, design, and implementation concepts. Each layer in the hierarchical structure consists of models and mappings are defined between layers. The significance of their work is incorporating the three aspects, technology (MDA's standards), business (EA), and information (LST's integration of layers) to provide a sound solution for model transformation.

2.2 Abstraction decomposition

The second approach is an attempt that tries to decompose the real-world problem into small pieces and derives the solution from the decomposed problems. Decomposition is also called divide and conquers, which is a word originated from ancient Latin saying, for referring a strategy by breaking big problem into small ones in order to manage one small problem at a time. This is a very general approach that has long been used to solve problems of many different domains, from mathematical proving [42] to computer hardware design [43]. The meaning of this approach to software evolution is the ease of evolution spotting. *Evolution spotting* is an action of finding related parts that should be changed as well when one specific part is changing (or changed). Since the problem, the solution, and the mapping between them are decomposed into smaller parts, it is easier to look up all parts that should be changed. However, it is important to choose abstractions for decomposition when the problem and the solution are at different worlds. This is the characteristic of E-type systems. We need a good abstraction (or abstractions) that can express the real-world and the computer world.

Two works that apply the idea of divide and conquer is reviewed here, one is Multi-Dimensional Separation of Concerns (MDSC) [44] and the other is Feature-Oriented Programming (FOP) [45]. These have two distinct choices of abstractions for decomposition. In MDSC, Clarke, Harrison, Ossher, and Tarr recognize the necessary of separating different types of concerns (features, business rules, objects etc) within programs. After these concerns are identified, programs can be composed and evolved as concerns change. They propose using different paradigms, where each paradigm is suitable for describing a single type of concerns. Conversely, FOP uses the one-single abstraction, called feature, for decomposing and composing different types of concerns. In FOP, software evolution of a program family can be incrementally synthesized [45] and evolved [46] from small features. Although they have different choices of abstraction representation, generally they both recognize the importance of choosing abstractions that are more closed to the problem domain. This is different from the functional decomposition [47]. This

approach concerns the solution provided by the external and internal functions of a system more than the other side (problem) of software development.

2.3 Traceability management

Finally, an approach that attempts to intuitively record every related part of any artifact is described. This approach is usually called traceability, which is a technique of linking different artifacts that produced during the process of software development, such as business cases, requirements, design relational, detailed design, code, documentation, and test cases. By recording how an item in an artifact is originated from other artifacts, it is possible to navigate to the artifacts that need to change when an artifact is changed. Ranging from using pen and paper to software support, it can provide an easy and powerful approach for maintaining the consistency between design and implementation. But tool support and automation of this technique is important for practical application.

Alves-Foss et. al. [48] describes a framework that represents design and implementation artifacts in XML. Xlink, a technology that provides the ability of linking between different XML documents, is used for providing traceability between XML documents transformed from UML design model and Java code. They use XSLT to transform the tracing Xlink to HTML document that provides hyperlinks between design specification and code. In [49], Anderson et. al. introduce an automatic approach for creating and maintaining traceability between different types of artifacts. Their work, based on the concept of information integration, defines different steps for managing tracing information and provides a conceptual framework consists of different entities for maintaining different types of artifacts in an uniform format. Similar to the approach of Alves-Foss et al, they also implement their framework by using XML, Xink, and XSLT. Different from our approach, both of their work use common design and implementation representation, such as UML, for defining traceability.

Chapter 3 Basic Framework

This chapter describes the basic framework of RSD. The purpose of this framework is to help developers to capture realization-development knowledge they acquire in the development process. In this chapter, a modeling language for this purpose is described from Sections 3.1 to 3.4. Section 3.5 describes the two distinct but also related modeling scenarios when using this modeling language. One is *domain modeling* for creating PRUs of one domain. The other is *application modeling* for reusing PRUs in one specific project. Section 3.6 describes supplementary graphical nations for visualizing realization-development knowledge.

3.1 Responsibility modeling for realization-development knowledge

Responsibility modeling is the modeling approach we use to capture the realization development knowledge. The core concept of this approach is responsibilities. We use this concept for modeling abstractions of different worlds. A responsibility in this approach concerns a task that should be performed by an entity on a type of information. From this definition, we can also model relationships between entities, which are defined as the connections between responsibilities. One important characteristic of responsibility is its wide-range of description. It does not only describe responsibilities of human entities but also artificial entities, such as business documents, or computable entities, such as software objects. To maintain the uniformity of modeling, we use the same structure of responsibility to model different work of different worlds. At the same time, we also provide flexibility of modeling. We can use constraints to give more details to a responsibility.

Responsibility modeling groups responsibilities into four worlds, where each world corresponds to a stage in the business system development process. Figure 3-1 depicts this

process as a cycle. The two outer boxes depict the main roles (system users and developers) involving in the stages. Responsibilities of each world represent different types of abstractions. In the world of business processes, responsibilities represent the collaborative work that is performed by business actors and business data that is processed by business actors. For example, a sales staff (a business actor) creates (work) a sales order (business data) for recording a purchase of goods or services (business data), and queries (work) the inventory of goods (business data) stocking for a customer (a business actor). In the world of user requirements, responsibilities represent the work that should be performed by the target system and the information that should be processed by the target software system. For example, the target software system performs a series of calculation and data accessing logic for automating the processing of a purchase that is inputted by a sales staff. In the world of software design, responsibilities represent the collaborative work of programming modules of the target system. For example, two software objects, one takes the responsibility to manage the data model of a purchase and the other is to access database, collaborate together for processing the electronic record of a purchase. In the world of program design, responsibilities represent the programming constructs that are used to give instructions to machines.

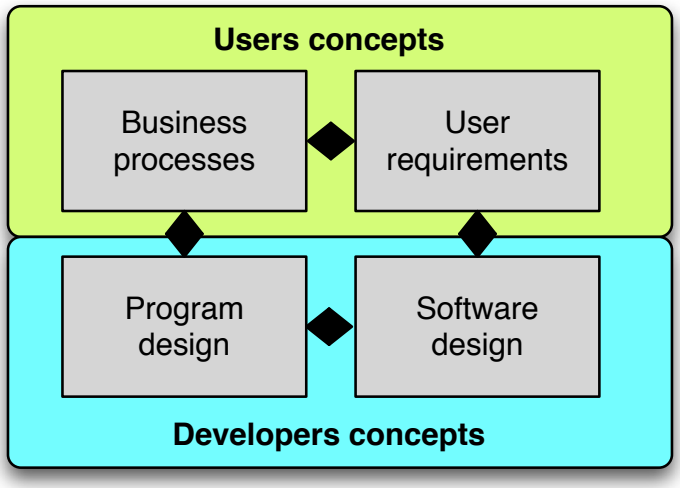


Figure 3-1. The cycle of four stages

To capture these different concepts in a well-formedness form, a metamodel is created that defines meta-constructs for describing these concepts. This metamodel is augmented by Object-Constraint Language (OCL) [50] that defines the detailed semantics these meta-constructs. One thing should be noticed is that this metamodel is defined within the framework, which implies it is specifically defined for the business domain. This metamodel includes three parts. Section 3.2 is the first part that defines the essential modeling elements. Section 3.3 is the second part that defines the modeling elements related to parameterized realization units (PRU). Section 3.4 is the third part that defines the modeling elements for managing other elements.

3.2 Essential modeling elements

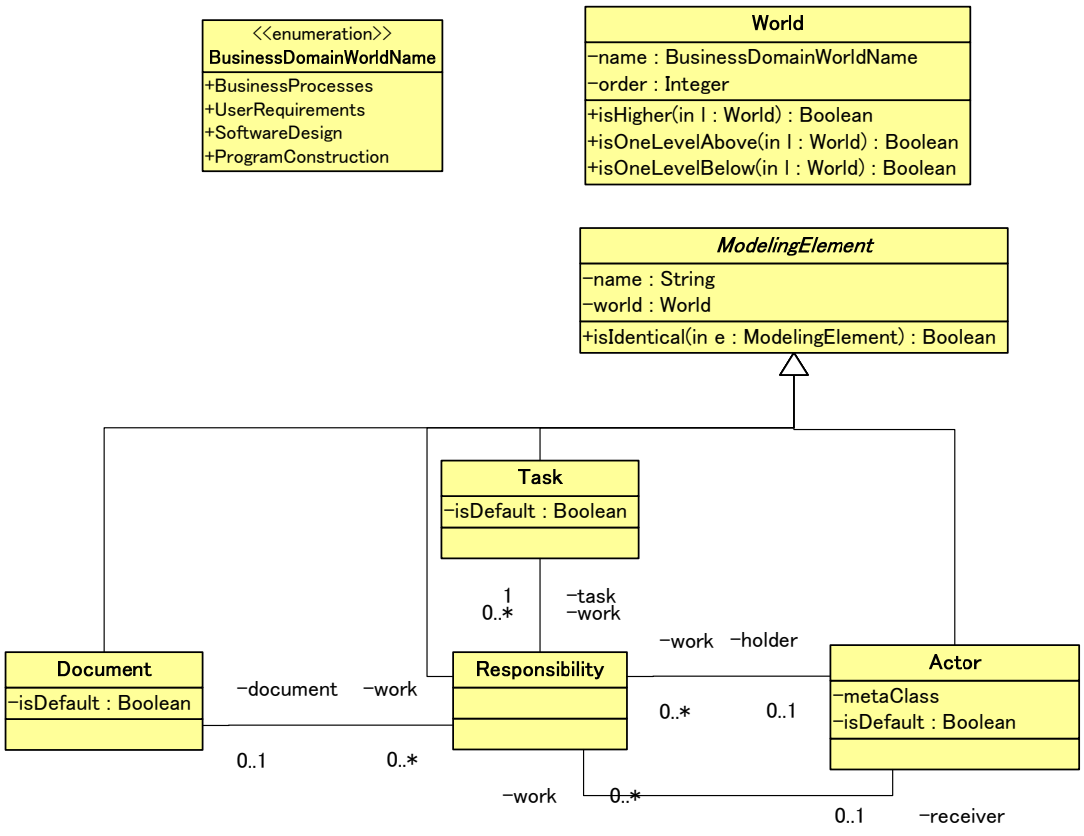


Figure 3-2 shows the metamodel for the essential meta-constructs. The essential meta-constructs, which include **`ModelingElement`**, **`Responsibility`**, **`Task`**, **`Actor`**, and **`Document`**, are described separately in the following sections.

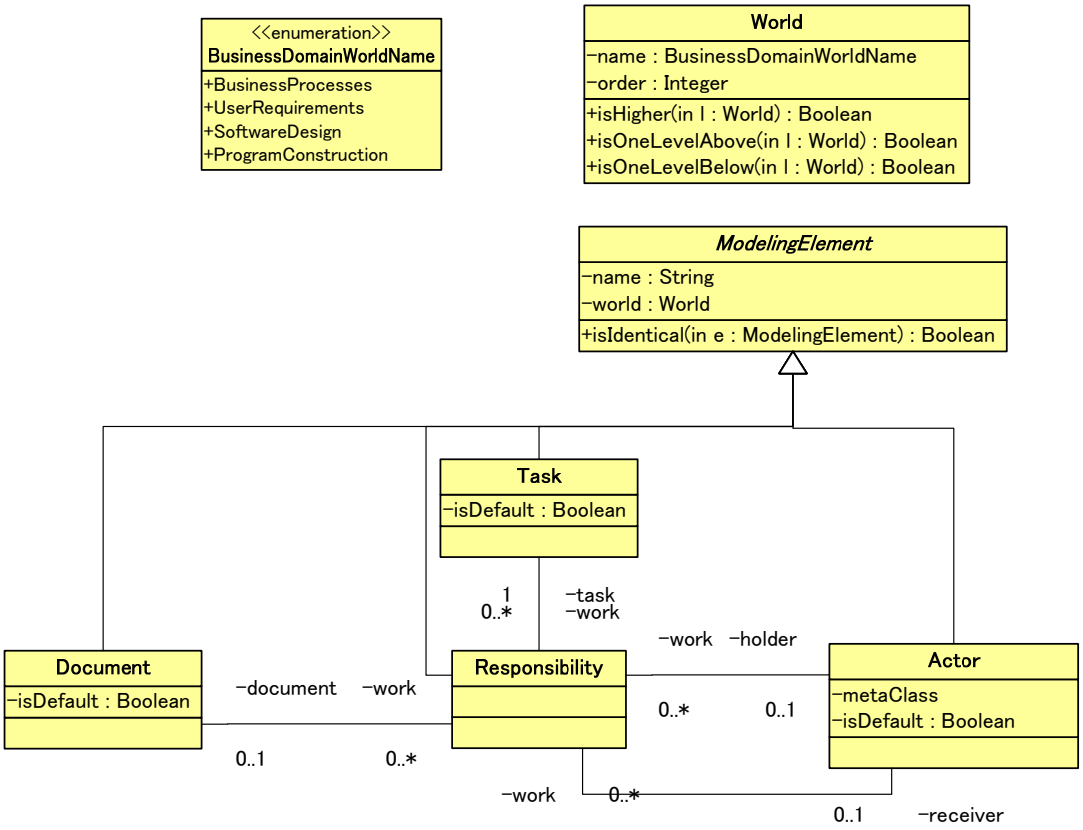


Figure 3-2. Metamodel of responsibility

3.2.1 ModelingElement

`ModelingElement` models the basic construct that can be extended. They define two attributes; `name` and `world` that can be inherit by other constructs, i.e. **`Actor`**, **`Concept`**, **`Document`**, **`Responsibility`**, and **`Task`**. **`ModelingElement`** has also a method **`isIdentical()`** to decide the identity of two instances of same type. This method should be overridden by subclasses of **`ModelingElement`** to define their specific logic.

[OCL-1] isIdentical()

The identity of two modeling elements is

```
context ModelingElement::isIdentical(e: ModelingElement) :
Boolean
body: self.world = e.world and
        self.name = e.name
```

3.2.2 Responsibility

`Responsibility` models a task that should be accomplished by an actor on some types of information. They are identified by name, and belonged to a world. They execute operations identify by task **`task`**. The operations perform on `document`. They are performed by actor **`holder`** and the results are sent to actor **`receiver`**.

[OCL-2] inTheSameLayer

The world of **`holder`**, **`receiver`**, **`task`**, and **`target`** should identical to the world of the responsibility. This can be expressed as an invariant of responsibility described by OCL. More details of the concept of world were already given in Section 3.1.

```
context Responsibility
inv inTheSameWorld: world = holder.world and
                        world = receiver.world and
                        world = task.world and
                        world = target.world
```

3.2.3 Task

`Task` models the operation of a responsibility performs. They are identified by `name`, and do not specify the actual behavior it performs. RSD focus on the modeling of the relationships of abstraction not the detailed specification of operations. Task has

isDefault that is used to indicate an instance of **Task** set in a responsibility is a default value or not. **Actor** and **Document** both have the same property. **isDefault** is used to define the property values of PRU.

3.2.4 Actor

Actor models an entity of a world that assumes a responsibility **work**. It can also model an entity that receives the performing results of a responsibility. Actors are identified by **name**, and can contain a number of attributes (not included in the essential metamodel). An actor could perform the work of a responsibility or receive the performing results of another responsibility.

3.2.5 Document

Document models a type of information that is processed by a responsibility work. They are identified by **name**, and can contain a number of attributes (not included in the essential metamodel).

3.3 Parameterized Realization unit (PRU)

RSD is different from other development methodologies. It focuses on realization relationship reusing, rather than implementation-based reusing. This approach is centered on a concept, called ***parameterized realization units (PRU)***, which models realization-development knowledge.

The main purpose of a PRU is to capture three types of relationships, which include (1) the realization of responsibilities between two worlds, (2) the collaboration between entities in the same world, (3) and the constraints that entities should follow. Each PRU provides a template for creating related abstractions between two worlds (i.e. stages) in one specific condition.

The idea of PRU can be understood better by the following example. When developers encounter a user requirement for displaying a list of open-orders, they create a

design, which may include several objects for realizing this user requirement. Without capturing this realization relationship between user requirements and software design in a model, developers may encounter the troubles of: (1) the necessary to locate where they have to make change in software design when this user requirement is evolved, (2) unaware of the accidentally change to the user requirement when the objects are changed, and (3) the worse is that they have to repeatedly re-create this relationship every time when they have to realize the same user requirement. They may create several designs with minor variation for every occurrence of this user requirement. The overall result is the inconstancy structure of software systems which are hard to be maintained. PRUs is to remedy all these troubles.

To clarify the semantics of meta-constructs that are used to create PRUs, a metamodel is provided in the basic framework. This is depicted in Figure 3-3. The following sections detail each meta-construct in the metamodel.

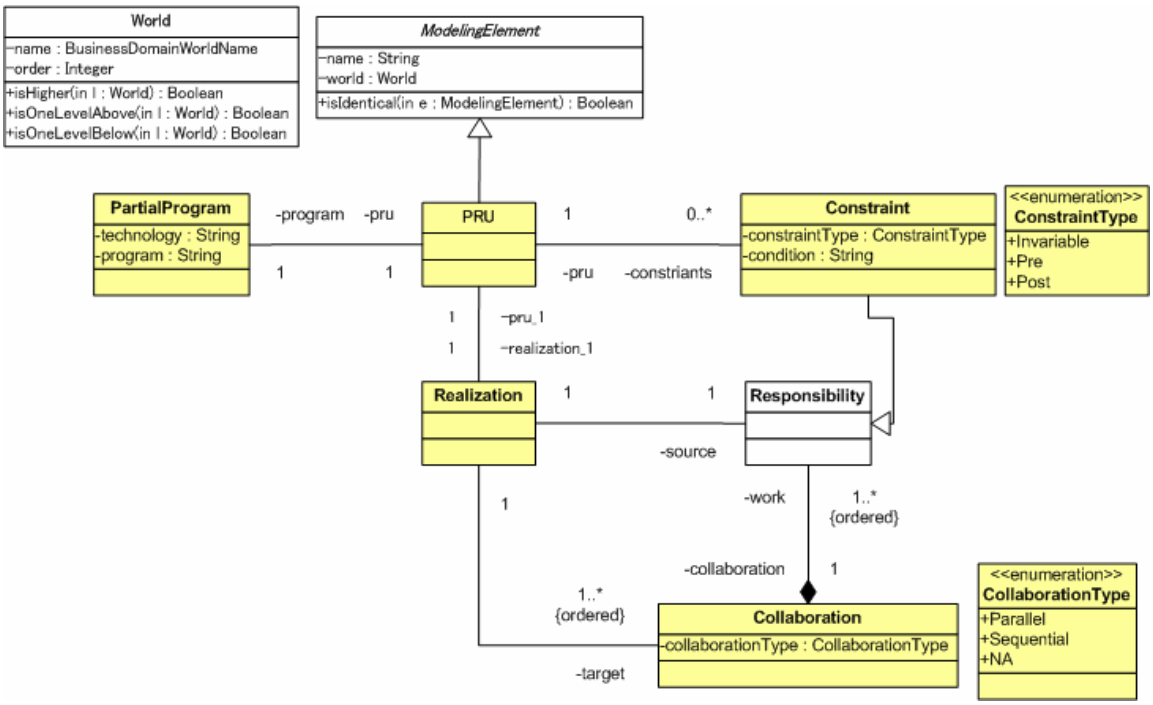


Figure 3-3. Metamodel of PRU

3.3.1 PRU

A PRU capture three types of relationships. To model the realization relationship between two worlds, a PRU contains a `realization`, which links two groups of abstractions. The first group is a responsibility `source.` The second group contains a collection of collaborative responsibilities `target`. At the same time, the responsibilities in the second group are connected together by a `collaboration` to model the collaboration between entities of the same world. Since there are four worlds defined in responsibility modeling, PRUs can be grouped into three categories, which describes the realization relationships between

- Business processes and user requirements
- User requirements and software design
- Software design and program construction

Finally, each PRU has `constraints`, which models some conditions that should be satisfied at (1) design and (2) implementation time.

For (1), it means that `target` should satisfy `source` when developers create a PRU. It applies to the PRU in the business-process and the user-requirements worlds. The semantics of constraints is propagated. Therefore, constraints in the earlier stage should be satisfied in the stages hereafter. For example, a constraint that states a nun-functional requirement of a business process should be realized by the user-requirements responsibilities that realize this business process, by software-design responsibilities that realize the user-requirements responsibilities, and by program-construction responsibilities that realize the software-design responsibilities.

For (2), it means that the implementation of the responsibilities of target should satisfy constraints. It only applies to the PRU in the software-design world because it defines the lowest-level of responsibilities that should be assumed by a program. A responsibility in the program-construction world has a `PartialProgram` attaching for defining the concrete implementation of this responsibility.

[OCL-3] .u ninn ninTwoWorlds

`source` and `target` should not belong to the same world.

```
context PRU
inv inTwoWorlds: source.world <> target.world
```

3.3.2 Realization

`Realization` models the links between two set of abstractions, `source` and `target`. `source` is a responsibility at one world and `target` is `collaborations`, which contains one or more **`Collaboration`**s, which in turn contains one or more **`Responsibilities`**. **`target`** is *ordered*, which means each collaboration of **`target`** is executed one by one (sequentially).

[OCL-4] realizeBetweenTwoWorlds

`source` should be in a world below `target`.

```
context Realization
inv realizeBetweenTwoWorlds:
                source.world.isOneLevelBelow(target.world)
```

3.3.3 Collaboration

`Collaboration` models the work that should be accomplished by one or more responsibilities, where each responsibility is a part of this work. Therefore, a collaboration can be conceptually considered as a *bigger* responsibility with many smaller responsibilities. Each collaboration has `collaborationType` indicating the execution type of the containing responsibilities. Therefore, responsibilities within one `collaboration` can be `Parallel`, `Sequential`, and `NA` (i.e. unknown). When a collaboration relationship is parallel, its contained responsibilities finishing their work at the same time. Conversely, when the type is sequential, its contained responsibilities finish their work one by one (sequentially).

[OCL-5] collaborateAtTheSameWorld

The contained responsibilities of a collaboration should be at the same world.

```
context Collaboration
inv collaborateAtTheSameWorld: work.world.name =
BusinessDomainWorldName.BusinessProcesses or
BusinessDomainWorldName.UserRequirements or
BusinessDomainWorldName.SoftwareDesign or
BusinessDomainWorldName.ProgramConstruction
```

3.3.4 Constraints

`Constraints` is extended from **`Responsibility`**. We consider **`Constraints`** is also a responsibility that should assumed by entities. The difference between a constraint and a normal responsibility is their application scope. A normal responsibility is a piece of a work performing by an entity. A constraint should be followed by all entities that are restricted by this constraint. For example, all business actors that involving in a business process. Therefore, it is suitable to define wider-scope requirements, such as the implementation technology of a target system or non-functional requirements of a target system.

`Constraints` are identified by name. `constraintsType` specifies the types of conditions, i.e. `Invariable`, `Pre` (i.e. pre-condition), and `Post` (i.e. post-condition). `condition` is the contents of a constraint. `condition` can be assigned by using any type of languages, e.g. OCL or natural language. RSD does not confine to any specific constraint language.

The types of a constraint are various. It can be a domain constraint which specifies an additional condition in terms of domain-specific concepts. For example, a constraint confines that a business-process responsibility should only list open orders. It can be a non-functional constraint which specifies non-behavioral condition. For example, a constraint confines that the query of all open orders should be completed within three seconds.

A constraint of a business-processes PRU (i.e. a PRU belongs to the business-processes world of which `world` is `BusinessProcess`) represents that the design of a business-processes PRU, including the `source` responsibility and `target` responsibilities of a collaboration, should satisfy this constraint. Therefore, the design of user-requirements responsibilities of this PRU does not only realize the work of the `source` responsibility but also confine to this constraint at the same time.

A constraint of a user-requirements PRU (i.e. a PRU belongs to the user-requirements world of which `world` is `UserRequirements`) represents that the design of a user-requirements PRU, including the `source` responsibility and `target` responsibilities of a collaboration, should satisfy this constraint. Therefore, the design of software-design responsibilities of this PRU does not only realize the work of the `source` responsibility but also confine to this constraint at the same time.

A constraint of a software-design PRU (i.e. a PRU belongs to the software-design world of which `world` is `SoftwareDesign`) represents that the design of a software-design PRU, including the `source` responsibility and `target` responsibilities of a collaboration, should satisfy this constraint. Therefore, the design of program-construction responsibilities of this PRU does not only realize the work of the `source` responsibility but also confine to this constraint at the same time.

One important thing should be noticed is that these properties to detail the design of PRU, such as `collaborationType` of **`Collaboration`**, `constraintsType` of **`Constraints`**, `attributes` of **`Document`** and **`Actor`**, are simply a mechanism for developers to record their design. They are not significant to RSD. That is, RSD and its implementation do not take the semantics of the values of these properties into consideration when evolving a software system. They are only used to for selecting a PRU for reusing. More details of PRU selection will be revealed lately in Section 4.4.2. But it will be an interesting extension as our future work.

3.4 Management of modeling elements

In order to manage PRUs created by developers, RSD provides the following meta-constructs. Figure 3-4 and Figure 3-5are the metamodel diagram.

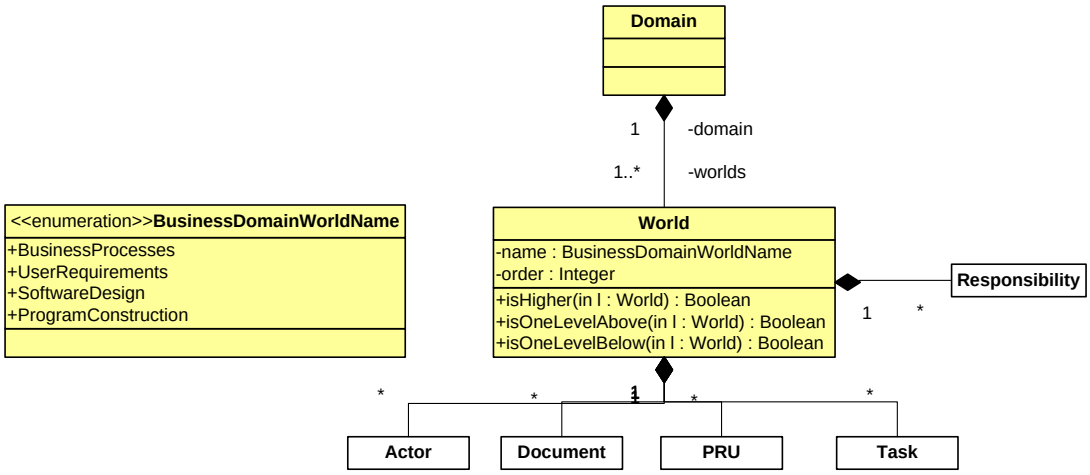


Figure 3-4. Metamodel of domain.

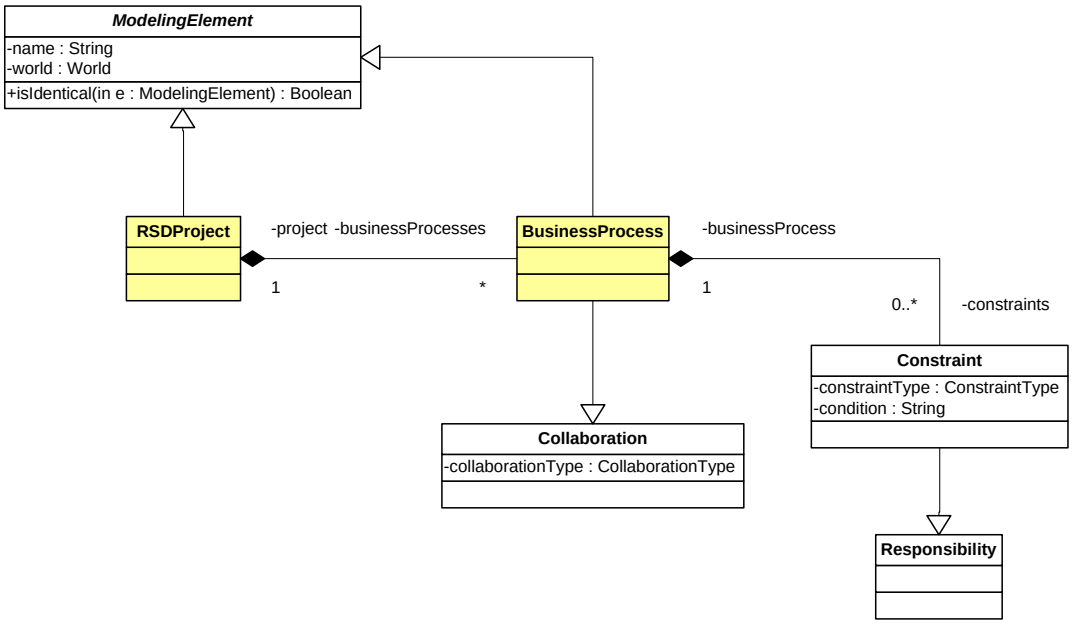


Figure 3-5. Metamodel of RSDProject

3.4.1 Domain

Domain models a container that can hold **World** (Section 3.4.2). Basically, the number and types of worlds are different for different domain. In this research work, there are four worlds defined, i.e. BusunessProcess, UserRequirements, SoftwareDesign, and ProgramConstruction, which are defined by **BusinessDomainWorldName**.

3.4.2 World

A **World** models a container that contains **Actor**, **Document**, **Task**, **Responsibility**, and **PRU** conceived in the corresponding stage.

3.4.3 RSDProject

A **RSDProject** models a container that contains business processes businessProcesses of one specific project. Different from **Domain** that contains abstractions to one domain, **RSDProject** contains abstractions that are specific to one single project.

3.4.4 BusinessProcess

A **BusinessProcess** models a container that contains one or many business-processes responsibilities businessProcessResponsibilities. It also models the collaboration of responsibilities. Therefore, it is extended from **Collaboration**. It is restricted by constraints, consequently all other responsibilities that realize this business process should honor this constraints.

3.5 Modeling Process

Sections 3.1 to 3.4 detailed the modeling language for capturing realization-development knowledge. This section describes how to use this modeling language in software development process.

The creation of RSD meta-constructs can be discussed from two modeling scenarios. The first is the modeling of abstractions that belong to one domain. The second is the modeling of abstractions that belong to one single project. The distinction between these two scenarios is clear. The first scenario is *domain modeling*. In domain modeling, developers create PRUs that can be reused for every project that belonging to the domain. The second scenario is *application modeling*. In application modeling, developers create business-process responsibilities and reuse PRUs created in the domain modeling for the creation/evolution of a program which realizes the business-process responsibilities. At the same time, the experience gained in application modeling also provides feedback to evolve PRUs. This distinction is similar to product-line development [51, 52]. It is depicted in Figure 3-6. Table 3-1 summaries the occurrence of meta-construct creation in these two scenarios. To read this table, the meta-construct that has marked the symbol ✓ means it is created in that scenario, otherwise, the symbol ✗. Basically, those constructs related to PRU are created in the domain modeling. Otherwise, they are the application modeling.

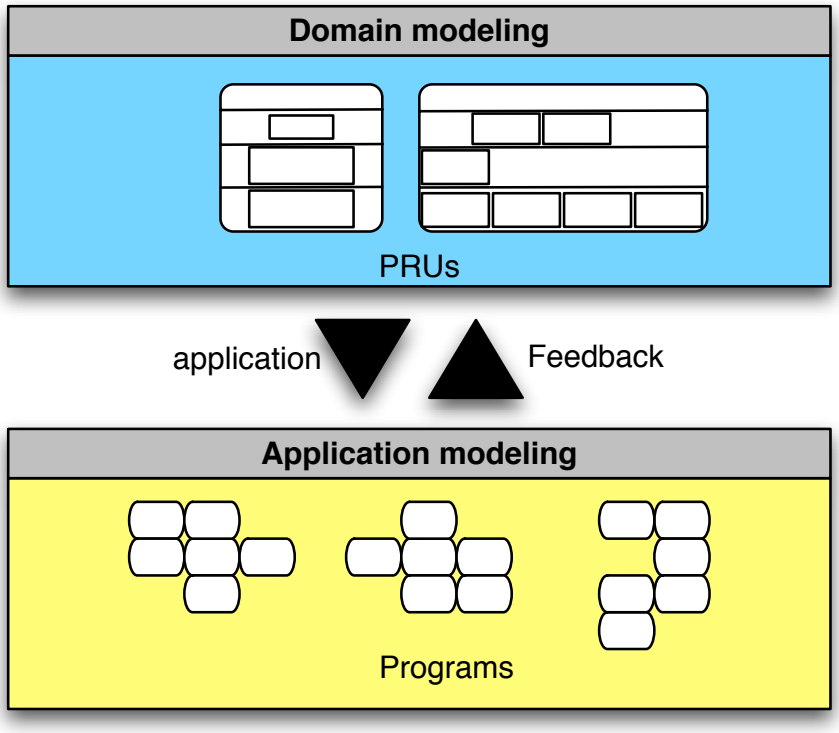


Figure 3-6. Domain and application modeling

Table 3-1. Modeling scenarios of meta-constructs creation

Meta-construct	Domain Modeling	Application Modeling
`Responsibility`	✓	✓
`Task`	✓	✓
`Actor`	✓	✓
`Document`	✓	✓
`Realization`	✓	×
`Collaboration`	✓	×

Meta-construct	Domain Modeling	Application Modeling
Constraints	✓	✓
Domain	✓	×
World	✓	✓
RSDProject	×	✓
BusinessProcess	×	✓

3.6 Graphical notations

3.6.1 PRU

Figure 3-7 shows an example of the graphical notation for modeling a **PRU**. A PRU is diagramed as a rounded rectangle. It has four compartments. From top to bottom, they are `world` and `name` of the PRU, `constraints`, `source` and `target` of responsibilities. A responsibility is diagramed as a rectangle. Its properties are placed from top to bottom as `world`, `task`, `document`, and `holder` and `receiver`. `holder` and `receiver` are surround by the symbol [ ], and are connected by the symbol →. There is no explicit visual modeling of `constraints`, `realization`, and `collaboration`. Instead, they are defined within different compartments.

It can be noticed that the responsibilities of Figure 3-7 (a) and Figure 3-7 (b) are different. In (a), `document` of each responsibility of `source` and `target` has a value specified. Conversely, in (b) `document` has `?` set instead, which indicates this property is *parameterized*. This parameterized property is the name *Parameterized* Realization Unit comes from. A parameterized property is a placeholder to accept different values. Therefore, this parameterized PRU can be instantiated with different documents, e.g. `SalesOrder`, `CustomerRecord`, etc., to create various similar realization design. More details of a

parameterized PRU are given in Section 4.4.1. Their real power will be revealed when the automatic implementation is described in Section 5.4.

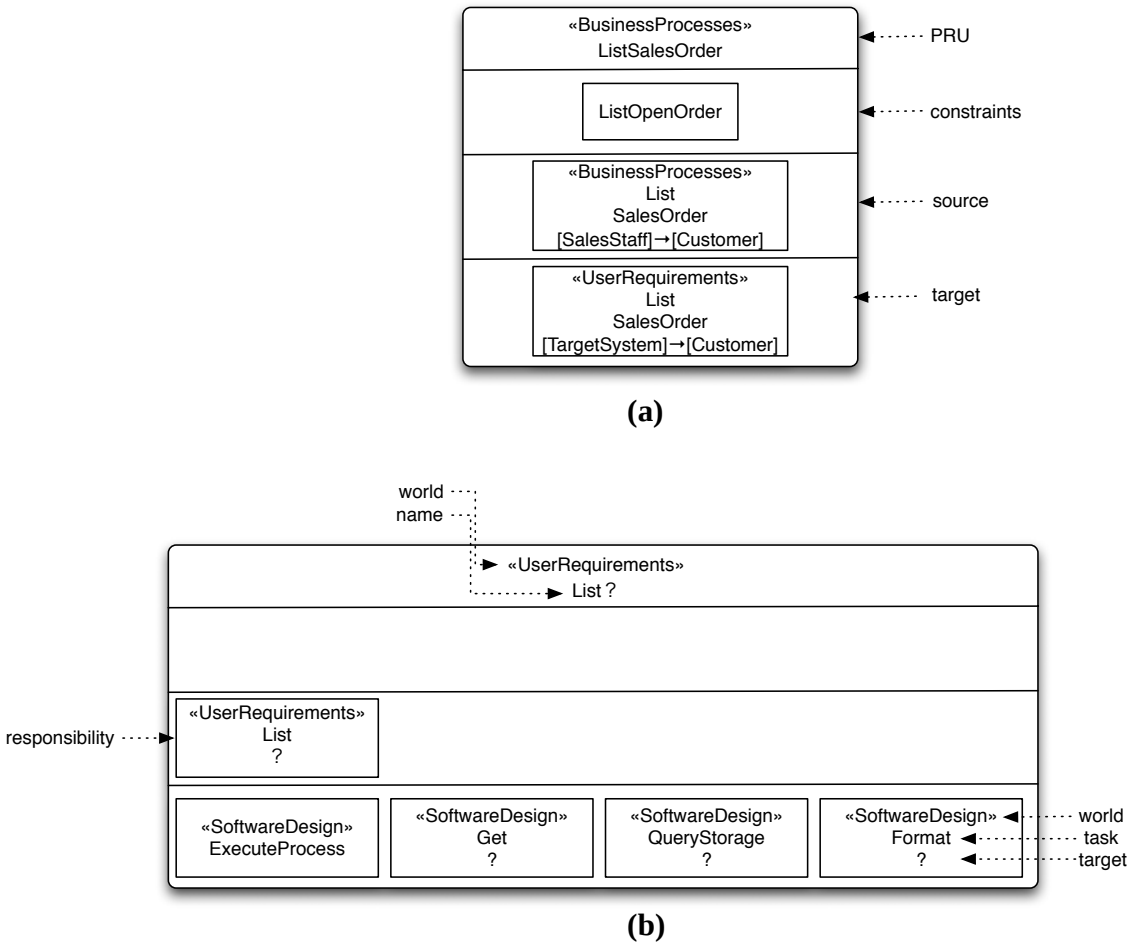


Figure 3-7. An example of graphical notation of PRU

3.6.2 Actor, Document, and Task

`Actor`, **`Document`**, and **`Task`** are modeled by using class notation of UML diagram. We use stereotype notation to distinguish the type of elements, i.e. **`«actor»`** for **`Actor`**, **`«document»`** for **`Document`**, and **`«task»`** for **`Task`**. Different from the modeling of PRU, the property `world` of these meta-constructs is not shown as stereotype but an attribute.

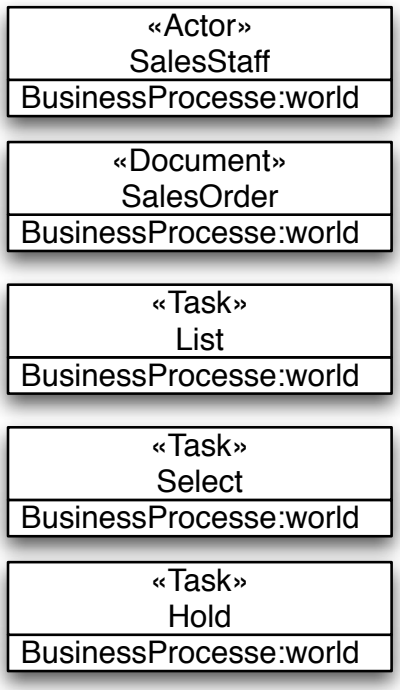


Figure 3-8. An example graphical notation for actor, document, and task

3.7 Stereotyping

Responsibility provides a concise concept for describing the task that should be accomplished by an entity of different worlds. The collaboration between entities provides another concise concept for describing how a "bigger" task is accomplished by multiple entities. However, one obvious problem that will rise is that the design of responsibilities and their collaboration should be created in a structured way. It is because that they also represent the design of a system. A structured design of responsibilities and their collaboration will bring a structured design of system.

To avoid an ad hoc design, we introduce another concept called stereotyping. Each responsibility of software design world will have a property stereotype, which can be one

of six values. Stereotypes characterize each responsibility with a specific role. Responsibilities with the same role exhibit same kind of work. The concept of stereotypes is firstly introduced in Responsibility-Driven Design (RDD) for helping developers design responsibilities and objects [5]. By stereotypes, it is easier for developers to create responsibilities.

In our research, we extend the application scope from finding responsibilities to constrain the communication path among object. That is, objects of certain stereotype will only communicate with objects of certain stereotypes. Following the original proposal of the six stereotypes, each responsibility of the software layer is characterized by one of the six stereotypes. They are list as follows:

- Holding information - know something
- Structuring - manage a set of structured objects
- Providing service - do something upon request
- Coordinating - reacts to events
- Controlling - Decide the process upon some criteria
- Interfacing - Process external request

We should decompose responsibilities based on these stereotypes. We should define a logical collaboration that require objects that commanding the work of other objects and taking different actions for different conditions. There are objects that interpret messages coming external to the target software system. The commanding objects decide the actions to take according the messages interpreting by these objects. They will also ask objects that hold information to pass information to other objects that providing services to process

information. At the same time, there should be objects that help the commanding objects managing those objects providing information and services. By these managing objects, the commanding objects can effectively retrieve the information they need and find the services they desire. When decomposing responsibilities from the system layer, developers follow the communication paths described above to design the work of responsibilities and the collaboration of responsibilities.

3.8 Summary

This chapter describes the basic framework of RSD. This framework helps developers to capture realization-development knowledge they acquire in the development process. A modeling language for capturing the realization-development knowledge is described. We describe its metamodel, which clarifies the semantics of meta-constructs, and its graphical notations, which is for visualization. We also states two different modeling scenarios, domain modeling and application modeling. In the domain modeling, developers create PRUs, the reusable modules of realization-development knowledge, for one specific domain. In the application modeling, developers reuse PRUs for the construction/evolution of a program.

Chapter 4 Reusing realization-development knowledge

This chapter describes the usage of PRUs, i.e. the modularization of realization-development knowledge, in program construction/evolution. Section 4.1 firstly gives an overview of this usage. In Section 4.2, we describe how a program can be constructed by only using PRUs. In Section 4.3, we describe how a program constructed by this way is easier to be evolved than by using other approaches. Section 4.4 differentiates the difference between a *parameterized* PRU and one that is not. It also describes a set of rules for prioritizing properties for PRU selection. After the introduction of how to construct/evolve a program by using PRUs, we clarify the reason of using responsibility for capturing realization-development knowledge.

4.1 Approach overview

We consider that for software evolution it is necessary to separate two type of activities, *evolution spotting* and *evolution action*. Evolution spotting is the activities to locate the part that should be changed in the development artifacts because of the changes of other parts. Evolution action is the activities to remove the existing artifacts to be replaced of by the new artifacts, and the integration of the new artifacts with the unchanged artifacts. To evaluate the ease of software evolution approach, we can see how these two activities are conducted.

Consider an example, where a sales staff has to add a new task in a sales-order-creation business process. This new task is to provide the real-time amount of stocking items to a customer, which was only collected every night. In order to realize this new task in the target system, developers have to locate the user requirements that realize the old business process, and then have to locate the objects that design to process the sales-order-

creation in database, and the concrete implementation of these objects. After these evolution spotting activities, developers have to take evolution actions to modify user requirements, software objects design, and program that realize the high-level abstractions. For a traditional multi-paradigm development methodology, developers have to switch their minds between textual representation and graphical representation, and between system behavior description, object structural and behavioral description, and various types of programming constructs, such as flow-control or variable definition.

RSD aims at remedying this issue. The assumption here is that when a program is constructed from the instantiation of PRUs then evolution spotting and evolution action is simplified as shown in Figure 4-1. This figure shows that how evolution spotting and evolution actions are simplified. For evolution spotting, instead of looking source code to locate a part that should be evolved, we can consult PRUs to match a PRU that is the old part instantiated from. Evolution spotting is simplified because the matched PRU has the information to find the related abstractions. For evolution actions, instead of manipulate source code to add a new part, we can consult PRUs to match a new PRU that can realize the new change. Evolution action is simplified because the repletion in software development is eliminated. In the figures, the old program is constructed by using the PRUs 1, 2, and 3. However, in the new program 1 and 3 are replaced by 4 and 5. If there is no suitable PRUs for realizing a new change, developers can create new PRUs for this new change which can also be reused for further development.

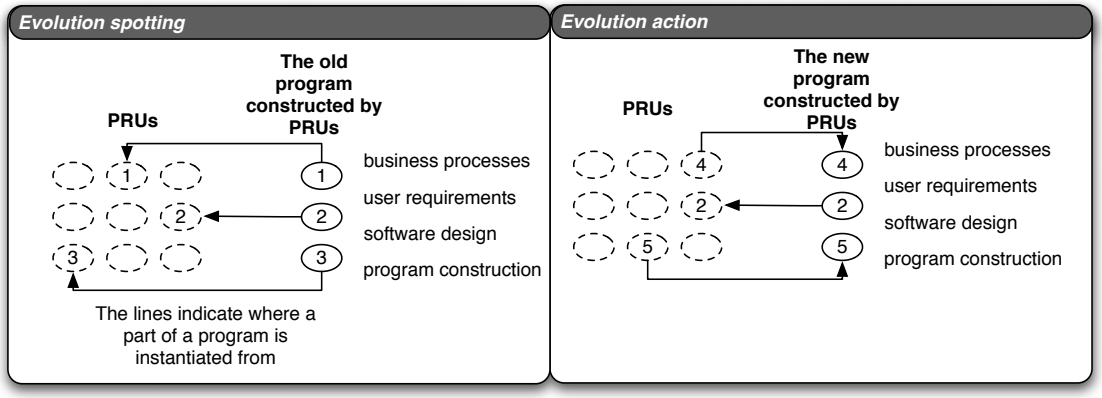


Figure 4-1. Software evolution helps by PRUs

Besides, since every PRU is described by a single-type paradigm, there is no necessary for developers to switch between different types of representation. Therefore, the following sections describe how a program that is constructed by PRUs is evolved by this approach.

4.2 Constructing a program by PRUs

A PRU contains responsibilities and the relationships of responsibilities. The combination of many instantiated PRUs represents all of the related responsibilities of a program that should be assumed. Since PRUs are categorized in terms of stages of software development process, to represent all of the responsibilities of one world (i.e. development stage in terms of RSD's terminology) that a program should assume, developers only have to find PRUs of that world. From the collaboration relationship, developers know how entities of that world collaborate together to assume their responsibilities. At the same time, from the realization relationship, developers know how these responsibilities are realized in the next stage. From this knowledge, we can find PRUs of each world and to instantiated all related responsibilities of a program. Finally, from the responsibilities of the program-construction world, there is a partial program attaching to a program-construction responsibility for combining into a complete program with other partial programs.

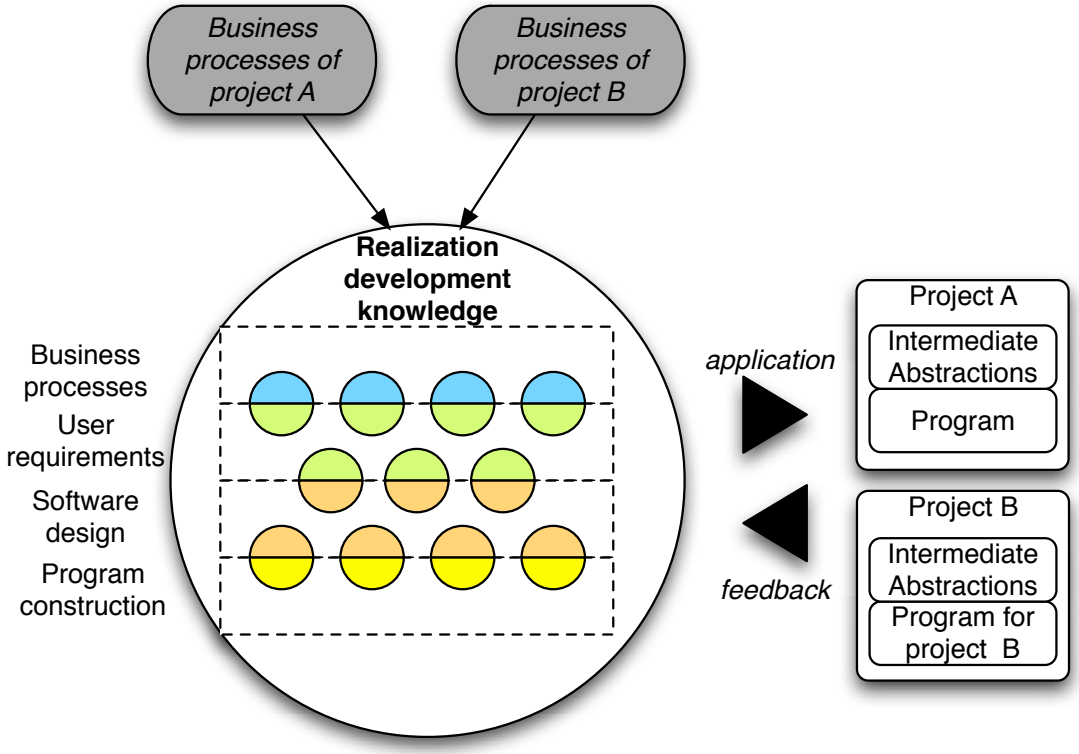


Figure 4-2 Creation of a RSD Program by using PRU. Intermediate abstractions are instantiated by using PRUs.

Figure 4-2 conceptually depicts creation process of a program by using PRUs. The central role of the program creation is played by the storage of the realization-development knowledge. While different projects use this central storage for creating intermediate abstractions and programs, each project also provides feedback to this central storage. Business processes are the input of this process. When the business processes of different projects that are represented as responsibilities are created, developers find intermediate abstractions by using PRUs that satisfy the input business-processes responsibilities, which in turn the user-requirements PRUs, and the software-design PRUs. Finally, from the software-design PRUs, the program-construction responsibilities and their attaching partial programs are known to create the two programs for the projects A and B. This central storage is live. It evolves at the same time. New PRUs are added to this when developers

acquire new realization-development knowledge. PRUs are removed from the central storage when some PRUs do not satisfy current development requirements.

The finding of PRUs can also be considered as a problem-to-solution process. The problem is an unrealized responsibility in one world and the solution is one or more responsibilities in another world that can be used to realize the unrealized responsibility. In this process, developers use this unrealized responsibility to find a PRU that knows how to realize this responsibility. The derived solution then becomes the new problem that should be solved. New PRUs for this world should then be found to realize these newly unrealized responsibilities. It should be noticed that this is different from the concept of design patterns [53] when a design pattern only describes the problem and the solution in the same world.

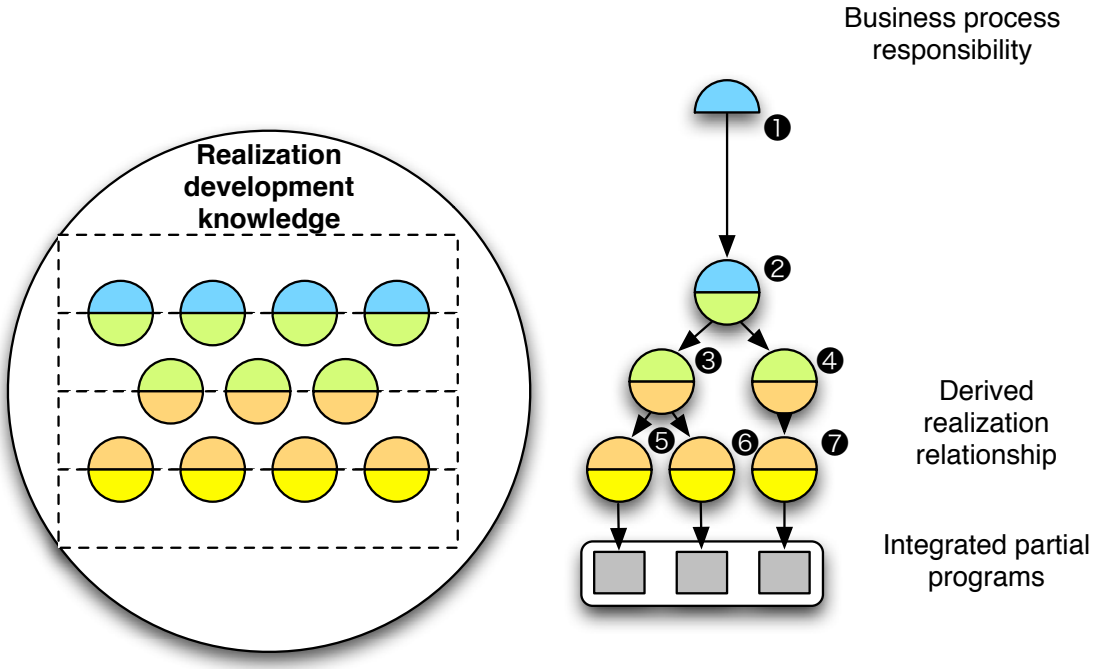


Figure 4-3. The details of the problem-solution process

More details of this process are shown in Figure 4-3. First, when a business-process responsibility is given (❶ in the figure), developers find a PRU (two ❷ in the figure) where

its `source` satisfies this responsibility. From target of ❷, one or more user-requirements responsibilities (two in this example) is created. For each of these two user-requirements responsibilities, developers instantiate one PRU for its realization (❸ and ❹ in the figure). This creates software-design responsibilities (a total of three in this example) for realizing the two user-requirements responsibilities. These software-design responsibilities represent the collaborative work that should be done by some programming modules, such as software objects, components, or HTML files. The details of the implementation will not be revealed until PRUs are found to realize these three software design responsibilities.

Two unrealized software-design responsibilities created by the PRU of ❸ are then realized by the instantiation of PRUs of ❺ and ❻. One unrealized software-design responsibility created by the PRU of ❹ is then realized by the instantiated PRU of ❼. These three PRUs that define the realization between software design and program construction have three partial programs attaching. The integrated program is the implementation for realizing the given business process and the intermediate abstractions found from these PRUs.

4.3 Evolving a program by PRUs

As shown in this example Figure 4-3, the central storage eliminates the repetition when realizing abstractions. When a program is constructed by this approach, we know (1) all intermediate abstractions between business processes and a concrete program, (2) all the connections between these abstractions and the program. (3) The knowledge that are used for deriving realization relationship. They are useful for evolving software within the scope of the three evolution scenarios, i.e. business-processes evolution, realization-development-knowledge evolution, and technology evolution. Figure 4-4 and Figure 4-5 depicts this.

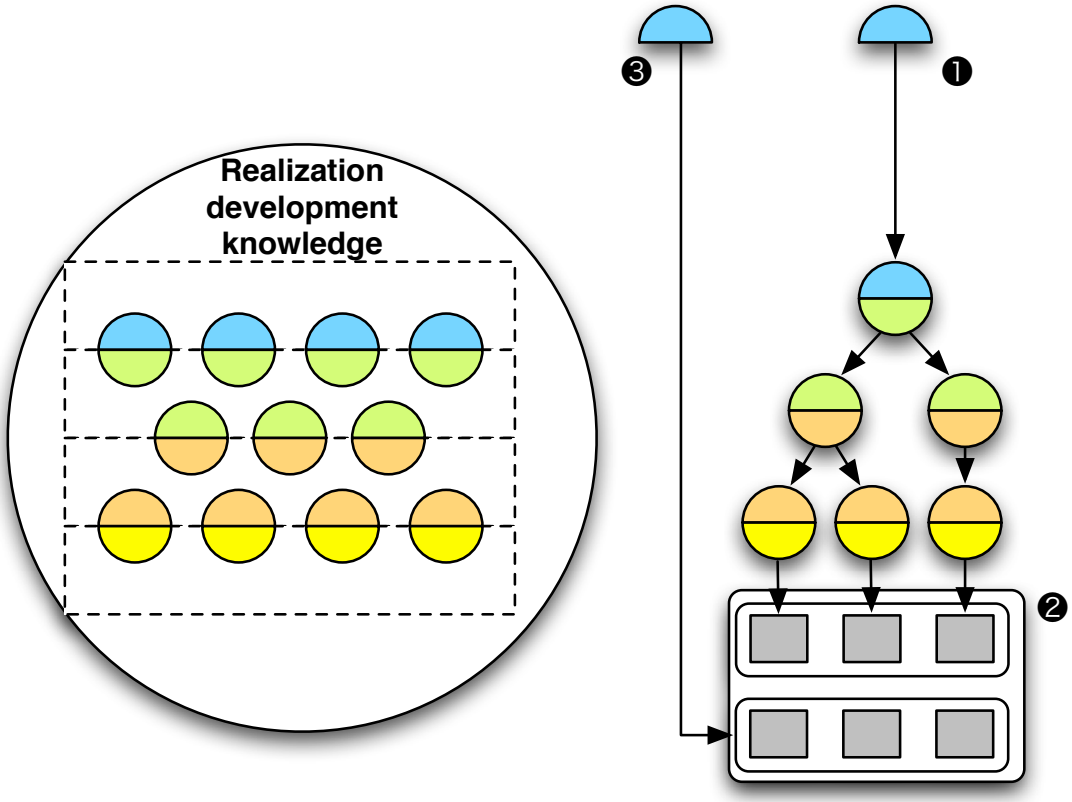


Figure 4-4. Evolution of a RSD program by using PRU

- Business-process evolution. When a new business-processes responsibility is added (see ❶ in Figure 4-4), the same process described in Section 4.2 is applied again. The newly derived partial programs (❷ in the figure) are integrated with the existing program that realizes a previously defined business-processes responsibility (❸ in the figure). This evolution scenario applies to the case where a new task is added to a previously realized business process.
- When a business-processes responsibility should be removed, the realization-development knowledge of PRUs is the source for locating the partial programs that realize this responsibility. By going through the same process described in Section 4.2, the partial programs can be found and removed.

- When the realization-development knowledge evolves, it is modeled as the removal of an existing PRU and the addition of a new PRU. The removal of a PRU represents that any RSD program generated before should be re-created in accordance to the newest development knowledge. The simplest approach is to use the same set of business processes responsibilities, apply the same process described in Section4.2, but use the new set of PRUs. Any RSD program can completely re-create. However, this is a very inefficient approach.

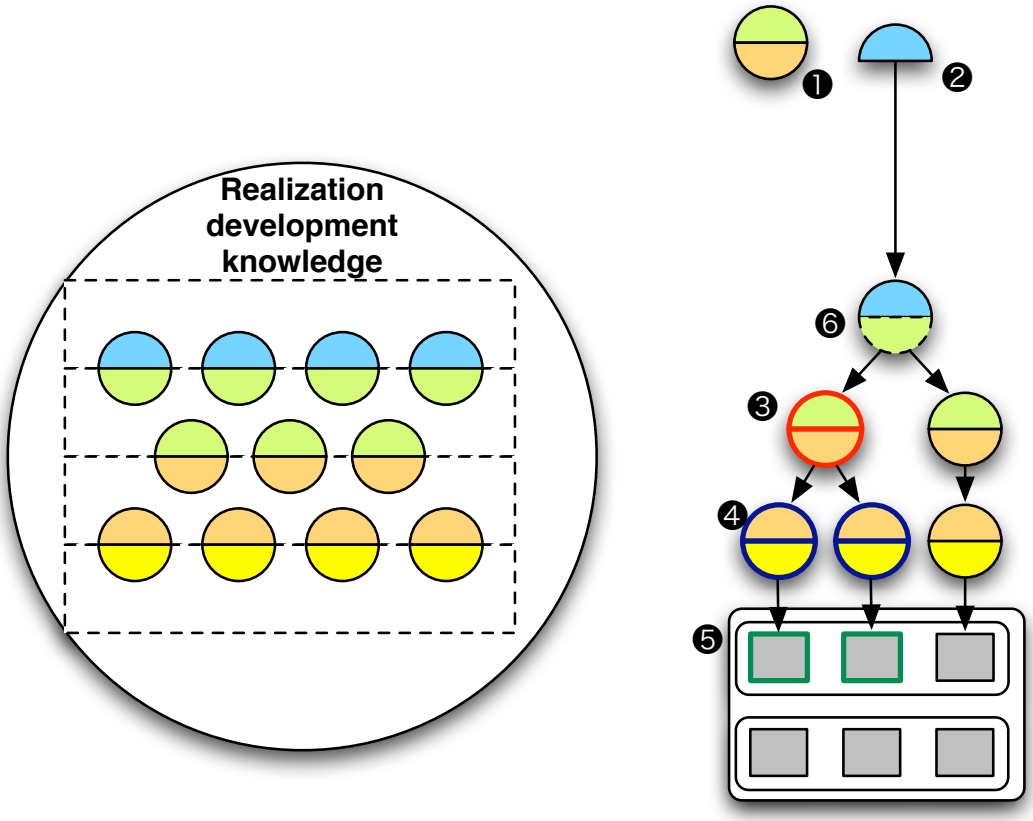


Figure 4-5. Realization-development knowledge evolution

Figure 4-5 depicts another algorithm. At start, the PRU that should be removed (❶) and the original set of business process responsibilities (❷) are given. The same process described in Section 4.2 is applied again. However, instead of finding those partial

programs for realization, the purpose of this process is to find the partial programs that were created from the old PRU (❶). When the realization relationship that is instantiated from the to-be-removed PRU is found (❸), other realization relationships (❹ and ❺) that were instantiated after this PRU are also located. Eventually, we find those partial programs (❻and❼) that were created for realizing the unchanged business-process responsibilities and the original development knowledge. These partial programs should be removed from the complete program because they are created based on the assumption of the to-be-removed PRU. This evolution spotting activity removes the partial programs of (❻and❼) from the complete program.

Now we have an unrealized user-requirements responsibility (❻ in Figure 4-5) to be realized. Developers have to use the same process described in Section 4.2 to find PRUs to realize this responsibility. If there is no PRU found in this process, they have to conceive new PRUs for realization.

One dilemma that we may face is the sharing of one PRU among several instances instantiated from that PRU. When a PRU should be removed, all instances that instantiated from this PRU should be removed as well. This situation may trigger a large-scale of program evolution because all instances of other PRUs for realizing this PRU should be removed as well. By using the modularization of development knowledge, we can solve such dilemma easily than other approaches. This can be understood better by Figure 4-6. It shows that the realizations for two business responsibilities both applied one shared PRU (❶ in the left and right). They can be easily evolved by the approach proposing in this research work.

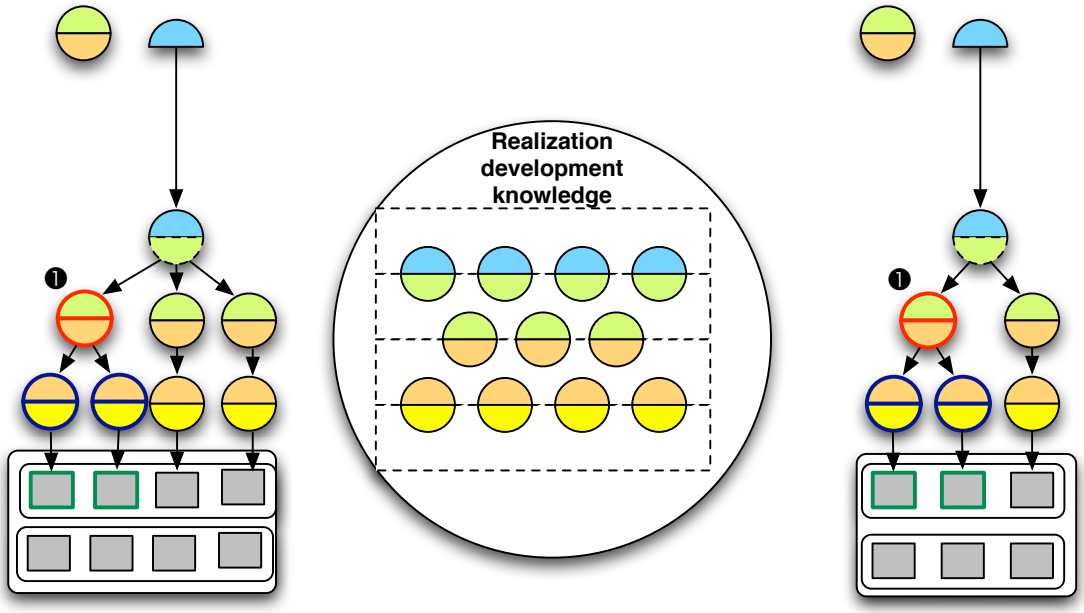


Figure 4-6. The evolution of two realizations shared one PRU.

4.4 Automating the reusing of development knowledge

The construction and evolution by using RPUs described so far can be automated. This automation is based on *parameterized* PRUs. A *parameterized* PRU has one or more properties set to no value when it is created, and are instantiated when a PRU is reused. It is used to create customizable realization relationship. We differentiate a *parameterized* PRU and one that is not in Section 4.4.1. The properties of a *parameterized* PRU represent different semantics. Therefore, the selection of a *parameterized* PRU for reusing should consider the semantics of properties. This consideration is called matching scheme, which is a set of rules that define which property should be matched first than others. This set of rules is described in Section 4.4.2.

4.4.1 Parameterized realization unit for knowledge reusing

Section 3.6.1 briefly introduced what a *parameterized* PRU is. This section gives more details of this matter. A PRU provides the customization of realization relationship creation. That is, from one single PRU, developers can create various similar concrete realization relationships by filling parameterized properties of responsibilities with different values. Consider the following business process,

- A shop owner selects desired features
- A customer selects purchasing items.
- A sales staff selects shipping items from stock

We represent each of these as a responsibility in Table 4-1.

Table 4-1. Example of business-processes responsibilities.

name	task	holder	documents	world
1a	Select	ShopOwner	Features	BusinessProcesses
2a	Select	Customer	Items	BusinessProcesses
3a	Select	SalesStaff	Items	BusinessProcesses

In order to realize these business-processes responsibilities, the following user requirements are defined.

- The target system has to list features for selection
- The target system has to list items for selection
- The target system has to list items for selection

Their responsibility representation is shown in Table 4-2.

Table 4-2. Example of user-requirements responsibilities

name	task	holder	documents	world
1b	ListForSelection	TargetSystem	Features	UserRequirements

name	task	holder	documents	world
2b	ListForSelection	TargetSystem	Items	UserRequirements
3b	ListForSelection	TargetSystem	Items	UserRequirements

The pairs of 1a and 1b, 2a and 2b, and 3a and 3b form three PRUs. 1a, 2a, and 3a are `source`. 1b, 2b, and 3b are `target`. Therefore, when there is an unrealized responsibility **na**, developers should create the responsibility **nb**. However, from these three units we can only create three concrete realization relationships. Actually, these three PRUs share a common structure. That is, they share one generic user-requirement responsibility of listing one type of information for selection. To model this generic responsibility, a very convenient approach is to parameterize the properties of a user-requirements responsibility by leaving some of its properties to be empty. These empty properties can be instantiated with different values for crating different responsibilities under different situations.

Both `source` and `target` of a PRU can be parameterized. The parameterized properties of `source` create an application condition that only those non-parameterized properties should be satisfied. The parameterized properties of `target` are placeholders to be filled with values for creating concrete responsibilities. Therefore, for this example we can create a new *parameterized* realization unit that can create abstractions of 1b, 2b, and 3b. This PRU is listed in Table 4-3.

Table 4-3. Example of a parameterized realization relationship

task	holder	world
`source`		
Select		BusinessProcesses
`target`		
ListForSelection	TargetSystem	UserRequirements

This PRU satisfies the given business process responsibilities 1a, 2a, and 3a, because their values in the properties `task` and `world` are identical to the same properties of the responsibility `source` of this PRU. The non-parameterized properties `task`, `holder`, and `world` of the parameterized responsibility in `target` are the default values when creating the user-requirements responsibility. The parameterized property `documents` of the responsibility target of this PRU are assigned the values from 1a, 2a, and 3a, that is, `Features`, `Items`, and `Items` respectively. The resulting user-requirements responsibilities are identical to 1b, 2b, and 3b in Table 4-2. This PUR is actually reused three times to cerate three realization relationships.

We call a PRU a ***base*** when its source responsibility has only `task` and `world` unparameterized, and has no constraint attached. It is because that it provides a most general situation for creating abstractions. Other PRUs that have identical values in `task` and `world` but different unparameterized properties and constraints defined are ***variation***s. Variations provide the chance for developers to fine tune the creation of a concrete realization relationship. For example, we can have one base PRU, which has `task` (=`Select`) and `world` (= `BusinessProcesses`), and a variation PRU, which has `task` (=`Select`), `world` (= `BusinessProcesses`), and `documents` (`Items`). Therefore, the creation of the realization relationship for selecting Items is different from the selection of other type of information.

One category of PRUs, i.e. the collection of PRUs that define the realization between the same pair of two different worlds, can be considered as the collection of many logical sub-categories. For every PRU in a logical sub-category, its `task` and `world` are assigned the same pair of values. For example, we may have a base PRU of which `task` is `LogError` and `world` is `UserRequirements`. There is also a variation of which `task` is `LogError`, `world` is `UserRequirements`, and documents is `InvalidAuthenticaiton`. These two PRUs, the base and the variation, form a logical sub-category for realizing any responsibility of which properties has at least `task` and `target` set to `LogError` and `UserRequirements`.

4.4.2 Matching scheme of PRU selection

A difficult situation created by the base and multiple variations of PRUs is that which PRU should be used from a sub-category. We summarize the matching scheme of PRU selection in Table 4-4. The left column is the name of property and the right column is the matching priority. A property with higher priority value is matched first.

Table 4-4. Matching scheme of PRUs

Property	Priority
world	1
task	1
documents	2
holder	3
receiver	4
constraints	*

In the selection process of PRU, the property with higher priority value is matched first. Therefore, for an unrealized responsibility, developers firstly pickup a PRU, and starting compare the values of `world` between the responsibility and the PRU, and the values of `task` between the responsibility and the PRU. If the values are identical, then developers go to the next property with the highest priority value. When they encounter a different value, they stop the matching process. They then check if the constraints of PRUs are satisfied. If it is, then this PRU is selected for realization. If it is not, then they pick another PRU and restart the process.

4.5 Why single-type paradigm modeling for abstraction and knowledge representation

After the description of RSD's basic approach for program construction/evolution, it is easier to explain why RSD is designed in this way. We clarify the reasons of choosing the

single-type paradigm for modeling and responsibilities for abstraction and knowledge representation in this section.

When considering the different choices of modeling approach, such as single-type modeling paradigm vs. multi-type modeling paradigm or general-purpose vs. domain-specific modeling approaches, we have two goals in mind.

- We want to narrow the gap between different contexts.
- We want to raise the abstraction level of program construction.

The first goal leads to the choice of single-type modeling paradigm. The second goal leads to the choice of domain-specific modeling approach. A single-type modeling paradigm simplifies the work of developers when creating connections between abstractions in different worlds. For example, when consider the realization of system behavior description by software objects, since the core concepts of these world, i.e., behavior of the target system and the behavior of objects, are different, developers have to switch their mind between two contexts and may have trouble to come up a solution. The inherent distance between these two worlds should be bridged by a single-type modeling paradigm, instead of the multi-type modeling paradigm which may aggravate the problem.

The current trend of implementation technologies is the variety of mechanisms. For example, in J2EE 1.4, we can use XML configuration files to designate the development or run-time properties of a system. However, in the newest version of 1.5, annotation is the preferred approach to accomplish the same task. This example also illustrates that not everything in code can be simply interpreted as objects or functions. It is better to provide a domain-specific language that provides another level of abstraction to isolate the evolution of programming models and to absorb the difference between different versions or different types of implementation technologies. The raising of abstraction level should also provide the necessary mechanism to link between high-level abstraction and their realization in a program.

However, these two goals are somehow contradict. It is because we need to find a single representation that does not only describe different upstream contexts in the

development process but also different implementation mechanisms used in a program. The decision we made is responsibility, a modeling paradigm we consider that does not only a well-recognized tool for designing objects [4, 5] or architecture but also a common concept existing in our real world. This characteristic is very useful when developers create higher-level abstractions in business processes and user requirements. We can model the work that should be done by business actors and the collaboration between business actors as responsibilities. We can also model the work of the target system that automates the work of business actors as responsibilities. It is even more useful when creating the design and implementation of a program. For example, representing the interaction between an EJB component and a XML configuration file as the responsibilities of two entities is more intuitive and integrated than as an association between an object and a file.

4.6 Summary

This chapter describes "what" have to do for constructing/evolving a program by using RSD. We firstly gave an overview of why a program created by using traditional development approach is inherently hard to be evolved. We then show that developers can create a program by deriving the intermediate abstractions from PRUs of different worlds. This can also be considered as a problem-to-solution process. The problem is an unrealized responsibility in one world and the solution is one or more responsibilities in another world that can be used to realize the unrealized responsibilities. After a program is constructed by using this approach, it contains the necessary information for evolution. We then describe how this information is used for three evolution scenarios: (1) business-process evolution, (2) knowledge evolution, and (3) technology evolution. We also detailed what a *parameterized* PRU is. A *parameterized* PRU has one or more properties set to no value when it is created, and is instantiated when it is used. It is used to create customizable realization relationship. Its power is used in automatic program construction/evolution and will be revealed in Chapter 5. Finally, we clarified the reasons of why RSD is designed in this way.

Chapter 5 Rule-based implementation

This chapter describes a rule-based implementation of RSD, which is called RSDTools. We describe the features it provides, the structure it is constructed, and the internal work of Jess [53], a Java-based rule-based engine, for automating the three evolution scenarios. Finally, we explain shortly of a possible implementation of version control in RSDTools.

Based on the basic framework, which contains metamodel and graphical notations, and the idea of reusing PRUs in software construction/evolution, a tool, called RSDTools, for automating software construction/evolution is implemented. By this tool, the complexity of three types of evolution scenarios can be managed by its automated capability.

Based on the third theory, RSDTools is implemented by integrating a rule engine for reasoning a program that realizes the given business processes. In this rule-based engine, two types of knowledge are encoded. The first type is PURs, which represent the realization-development knowledge. The second type is the matching scheme for selecting and instantiating a PRU. By the combination of these two, this tool achieves the research goal of realization-evolution automation.

5.1 Features of RSDTools

RSDToools provides the following features:

- Graphical modeling for both the domain modeling and the application modeling. In the domain modeling, developers design PRUs for reusing. In the application modeling, developers create business-processes responsibilities and reusing PRUs created in the domain modeling for constructing and evolving a program.
- Automatic Jess facts generation. Jess facts encode realization-development knowledge. This feature includes the generation of PRUs and unrealized business-process responsibilities to Jess facts.

- Jess rules for PRUs selection and instantiation implementation. Jess rules encode the matching scheme of PURs selection and instantiation. These Jess rules are pre-loaded onto Jess for automatic software evolution. These rules infer partial programs that will be used to realize business-process responsibilities created in the application modeling.

5.2 Structure of RSDTools

Figure 5-1 shows the overall structure of RSDTools. RSDTools is constituted of three major components. *Graphical modeling component* (GMC) provides a graphical environment for modeling realization-development knowledge and unrealized business processes. More specifically, developers use the concept of responsibility for creating the models of (1) the collection of the three categories of PURs, which includes those between business processes and user requirements, those between user requirements and software design, and those between software design and program design, and (2) the collection of business-processes responsibilities that should be realized. These two models follow the semantics described in Sections 3.2 to 3.4. We call the first model *PRU model* and the second *business process model* (BP model for short).

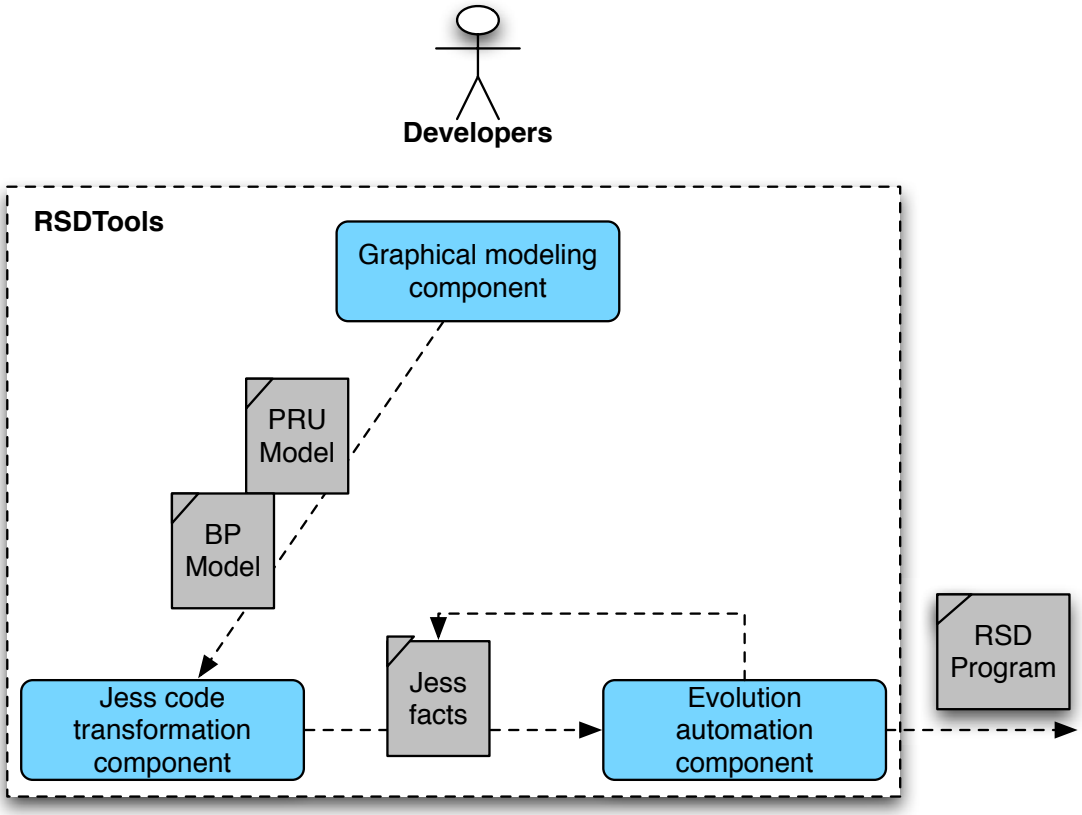


Figure 5-1. High-level structure of RSDTools

Jess code transformation component (JTC) provides the automatic transformation from the PRU model and BP model to Jess facts. *Evolution automation component* (EAC) is the core of this tool. It is the integration of Jess. This component contains Jess rules for selecting and instantiating PRUs. The input of this component is the Jess facts of PRU model and BP model, which are generated by JTC, and the output is the partial programs which is reasoning by the Jess rules for realizing the BP model.

RSDTools is based on several Eclipse technologies. We use Eclipse Modeling Framework (EMF) [55, 56], which is a modeling framework and code generation facilitate, to generate basic modeling implementation. Another technology is Graphical Editor Framework (GEF) [57]. We combine GEF and its supporting graphical drawing library, draw2D, with EMF for implementing Graphical modeling component (GMC). EMF

provides the functionality of creating the concept model and the responsibility models. GEF provides a graphical front-end for visually modeling. However, this combination is far from productivity. One major flaw of this combination is that synchronization between the metamodel created in EMF and the graphical representation created in GEF should be managed manually. Once there is any change in the metamodel, it is hard to make necessary changes in GEF. Finally, we adopt Graphical Modeling Framework (GMF) [58], which is another Eclipse technology. By GMF, graphical representation can be automatically created from the metamodel created in EMF.

5.3 Automatic Jess code generation

JTC transforms the PRU model and the BP model to Jess facts. Jess rules are preloaded onto Jess by our implementation. We describe the generated Jess facts of the PRU model and BP model in Section 5.3.1. We describe the preloaded Jess rules for construction/evolution in Section 5.3.2. We show the output results from Jess of this example in Appendix B.

5.3.1 Structure of modeling elements in Jess templates

Each Jess template defines a type of concepts. It is similar to classes in OOP. Jess facts are created from these templates. Therefore, Jess facts are similar to instances in OOP. Jess template should be defined by using `deftemplate`, `slot`, and `multislot` as the keywords. `deftemplate` starts a definition of a type of concepts. `slot` defines a single value property of the concept. `multislot` defines a multi-value property of the concept.

Figures 5-2 to 5-7 list Jess templates for **`Actor`**, **`Document`**, **`BusinessProcess`**, **`Responsibility`**, **`PRU`**, and **`Collaboration`**. **`Realization`** does not explicit transform as a concept (i.e. a Jess template), but is embedded as properties (i.e. `slot`) of **`Responsibility`**'s Jess template.

```
(deftemplate actor
    (slot id)
```

```
    (slot project)
    (slot name)
    (slot collection-name)
    (multislot propertyTypes)
    (multislot propertyNames)
    )
```

Figure 5-2. deftemplate **of** **Actor**

```
(deftemplate document
    (slot id)
    (slot project)
    (slot name)
    (slot collection-name)
    (multislot propertyTypes)
    (multislot propertyNames)
    )
```

Figure 5-3. **deftemplate** **of** **Document**

```
(deftemplate process
    (slot id)
    (slot project)
    (slot name)
    )
```

Figure 5-4. deftemplate **of** **BusinessProcoess**

```
(deftemplate responsibility
    (slot id)
    (slot project)
    (slot name)
    (slot from-pru)
    (slot world)
    (slot process)
    (slot task)
```

```
    (slot holder)
    (slot receiver)
    (slot document)
    ;; this slot keeps the ids of realized target
    (slot target-counter))
```

Figure 5-5. `deftemplate` **of** **`Responsibility`**

```
(deftemplate pru
    (slot id)
    (slot world)
    (slot name)
    (slot source)
    (slot target-count)
    )
```

Figure 5-6. `deftemplate` **of** **`PRU`**

```
(deftemplate collaboration
    (slot pru)
    (slot sequence)
    (slot target)
    )
```

Figure 5-7. `deftemplate` **of** **`Collaboration`**

5.3.2 Example of Jess facts

From Figures 5-8 to 5-12 , we list the examples of the generated Jess facts in the Business-MS system. These examples are generated for the scenario. Figure 5-8 lists Jess facts for two actors, `SalesPerson` and `Customer`. We generate one unique id number for each **`Actor`** in case of any possible duplicated name between two actors. Actors are general to one project. Therefore they can be shared between business processes of that project. In the example, these two actors belong to the project Business-MS.

```
(deffacts BPMS-Actors
```

```
    "BPMS Aactors"
    (actor (id 0)
        (project Business-MS)
        (name SalesPerson)
        (collection-name SalesPeople)
        (propertyTypes Department)
        (propertyNames String))
    (actor (id 1)
        (project Business-MS)
        (name Customer)
        (collection-name SalesPeople)
        (propertyTypes Credit)
        (propertyNames Money))
    )
```

Figure 5-8. Example of `Actor`'s Jess facts.

Figure 5-9 lists Jess facts for four documents, `SalesOrder`, `ItemCatalog`, `Item`, and `ShoppingCart`. We also generate one unique id number for each **`Document`** in case of any possible duplicated name between two documents. Documents are general to one project. Therefore they can also be shared between business processes of that project. In the example, these four actors belong to the project Business-MS.

```
(deffacts BPMS-Document
    "BPMS Documents"
    (document (id 0)
        (project Business-MS)
        (name SalesOrder)
        (collection-name SalesOrders)
        )
    (document (id 1)
        (project Business-MS)
        (name ItemCatalog)
```

```
        (collection-name SalesOrders)
        )
    (document (id 2)
        (project Business-MS)
        (name Item)
        (collection-name Items)
        )
    (document (id 3)
        (project Business-MS)
        (name ShoppingCart)
        (collection-name Items)
        )
    )
```

Figure 5-9. Example of `Document`'s Jess facts.

Figure 5-10 lists facts for the business process `CreateSalesOrder` we intend to realize. We also generate one unique id number for one business process.

```
(deffacts BPMS-process
    "BPMS Business Processes"
    (process (id 0)
        (project Business-MS)
        (name CreateSalesOrder))
    )
```

Figure 5-10. Example of `BusinessProcess`'s Jess facts.

Figure 5-11 lists Jess facts for business-processes responsibilities. They are the unrealized responsibilities that can be solved by our RSDTools implementation. Similar to Actor and Document, Responsibility is also given one unique id number.

```
(deffacts BPMS-CreateSalesOrder-Responsibilities
    "BPMS CreateSalesOrder Responsibilities"
    (responsibility (id 0)
```

```
    (project BPMS)
    (process 0)
    (task List)
    (holder SalesPerson)
    (document ItemCatalog)
    (target-counter -1))
(responsibility (id 1)
    (project BPMS)
    (process 0)
    (task Select)
    (holder Customer)
    (document Item)
    (target-counter -1))
(responsibility (id 2)
    (project BPMS)
    (process 0)
    (task Hold)
    (holder SalesPerson)
    (document ShoppingCart)
    (target-counter -1))
(responsibility (id 3)
    (project BPMS)
    (process 0)
    (task Checkout)
    (holder Customer)
    (document ShoppingCart)
    (target-counter -1))
(responsibility (id 4)
    (project BPMS)
    (process 0)
    (task Create)
```

```
        (holder SalesPerson)
        (document SalesOrder)
        (target-counter -1))
 )
```

Figure 5-11. Example of business-processes responsibility's Jess facts.

Figure 5-12 lists Jess facts for **PRU**s. There are two business-processes **PRU**s. It can be observed that there are two parts in these Jess facts. One defines the properties of PRUs. The other defines parameterized responsibilities `source` and `target`.

```
(deffacts Business-Process-World-PRUs
    "PRUs of the business process world"
    (pru (id 0)
        (world BusinessProceses)
        (name pru-0)
        (source 0)
        (target-count 1)
        )
    (collaboration
        (pru 0)
        (sequence 1)
        (target 1)
        )
    )

(deffacts Business-Processes-World-Responsibilities
    "source responsibilities and target responsibilities"
    (responsibility
        (id 0)
        (world BusinessProcesses)
        (task List)
        )
```

```
(responsibility
    (id 1)
    (world UserRequirements)
    (task List)
    )
)
```

Figure 5-12. Example of business-processes PRU's Jess facts.

5.3.3 Jess rules

We implement the selection and instantiation of PRUs for realizing abstractions of different stages as Jess rules. Jess rules are defined by the Jess **`defrule`** construct. The symbol **`=>`** separates the two parts of rules. Before this symbol is the IF-part and after this symbol is the THEN-part. The IF-part contains patterns that match Jess facts. Operations of the THEN-part are invoked when the patterns in the IF-part match some Jess facts. The operations usually made modification to Jess facts or call external Java programs.

We list one example of this implementation in Figure 5-13. This example select PRU between base and the variation with one target defined for realizing any unrealized responsibility. This rule applies to all three worlds, business-processes, user requirements, and software-design. In the IF-part, i.e. before the symbol `=>` this rule tries to select one PRU and one responsibility in target of that PRU. This responsibility in target can realize the unrealized responsibility. The THEN-part, i.e. after the symbol `=>`, instantiates (i.e. assert) this responsibility in target. This rule instantiates one responsibility at a time. When all responsibilities for realizing a unrealized responsibility are instantiated, another rule, which is shown in Figure 5-14 will remove this just-satisfied responsibility.

```
(defrule realize-unrealized-responsibility-use-base
    "This rule realize an unrealize business-processes
responsibilities"
    ;; match a constraint
    ;; ?c <- (constraint (id ?cnt-id) (lowLevel-counter nil))
```

```
    ;; an unrealized responsibility
    ?un-r <- (responsibility (id ?un-r-id)(project ?un-p-
id)(task ?un-r-task)
        (document ?un-r-target)(target-counter ?un-r-target-
counter&:(neq ?un-r-target-counter 0)))
    (or (and
            ;; match a PRU
            ?pru <- (pru (id ?pru-id)(source ?r-id-s)(target-
count ?r-target-count))
            ;; who's source and the concept both have same
task and same target
            (responsibility (id ?r-id-s)(world ?r-world-s)
(task ?un-r-task)
                (document ?r-target-s&:(eq  ?r-target-s ?un-
r-target))))
        ;; or
        (and
            ;; match a PRU
            ?pru <- (pru (id ?pru-id)(source ?r-id-s)(target-
count ?r-target-count))
            ;; who's source has no target set, i.e. base PRU.
            (responsibility (id ?r-id-s)(world ?r-world-s)
(task ?un-r-task)
                (document ?r-target-s&:(eq ?r-target-s nil))))
        )
    (collaboration (pru ?pru-id)(target ?r-id-t))
    ;; whose target is in the next stage
    (responsibility (id ?r-id-t)(world ?r-world-t&:(eq ?r-
world-t (next-world ?r-world-s)))
        (task ?r-task-t) (document ?r-target-t))
    ?done-target <- (done-target (source ?un-r-id) (done
$?done&:(not (member$ ?r-id-t ?done))))
    =>
    ;; get a new unique id for a respoonsibility
```

```
    (bind ?ed-r-id (get-responsibility-id))
    ;; add a new unrealized responsibility.
    (assert (responsibility (id ?ed-r-id)(from-pru ?pru-
id)(world ?r-world-t) (task ?r-task-t) (document ?r-target-
t)))
    ; decreae one of counter
    (modify ?un-r (target-counter (- ?un-r-target-counter 1)))
    (modify ?done-target (done (insert$ ?done 1 ?r-id-t))))
```

Figure 5-13. Example of Jess rules for selecting and instantiation abstractions.

In Figure 5-14, we list the rule to remove (i.e. retract) the responsibility that is satisfied by the instantiation of responsibilities by the rule in Figure 5-14. We use a special slot target-counter to record how many remaining responsibilities are necessary to realize this responsibility. Therefore, once this slot is set to 0, Jess will retract this just-satisfied responsibility.

```
(defrule retract-satisfied-responsibility
    "Retract a satisfied responsibility"
    ;; This rule should have highest priority
    (declare (salience 100))
    ?r <- (responsibility (target-counter 0))
    =>
    (retract ?r))
```

Figure 5-14. Jess rules for retract just-satisfied responsibility.

5.4 Automation of program construction/evolution

In the *Evolution automation component* (EAC), Jess rules are used to automate the construction/evolution of a program by matching between Jess facts of unrealized responsibilities of a project and Jess facts of PRUs of the business domain. Section 5.4.1 describes the internal work of EAC when constructing a RSD program. Sections 5.4.3 to 5.4.5 describe the internal work of EAC when evolving a RSD program.

5.4.1 RSD program construction

Figure 5-15 depicts the internal work of EAC when constructing a RSD program. When the Jess facts of a business-processes responsibility R1 asserted (i.e. added) to Jess by JTC, Jess consults Jess facts of PRUs. In the diagram, it shows that U1 is the PRU found because it specifies the realization of R1. From U1, the matching rules [1] generate the user-requirements responsibilities, which are the responsibilities R2 and R3, from the specification of U1 (in the right of Figure 21). After R2 and R3 are generated, the matching rules found that U2 and U3 are the PRUs that specifies the realization of the newly generated R2 and R3. This time, the matching rules will create the software-design responsibilities from the specifications of U2 and U3. Following the same manner, U4 is found for R4, and U5 and U6 are found for R5 and R6.

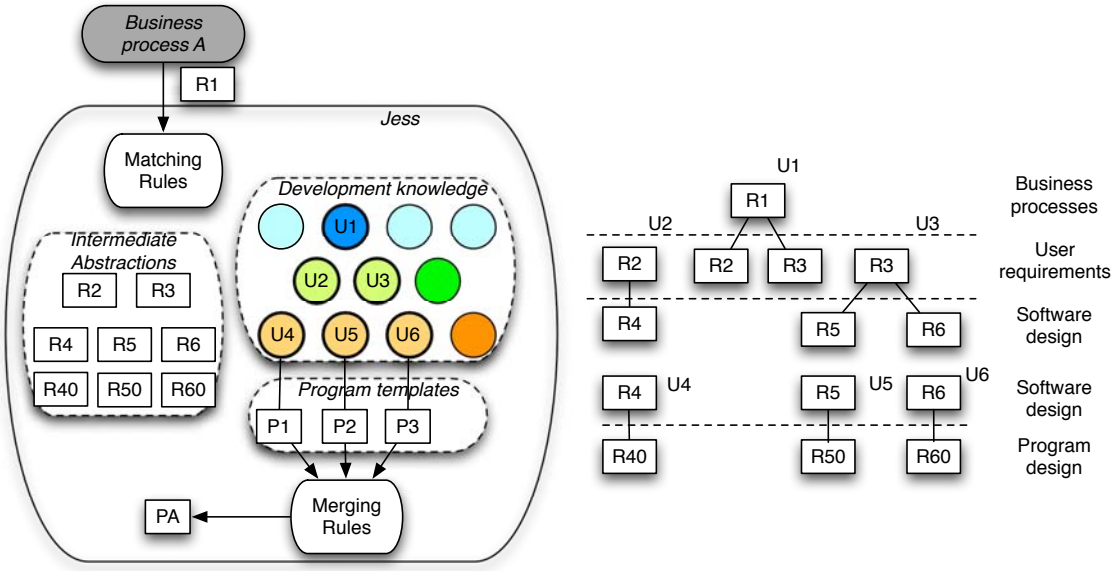


Figure 5-15. The internal work of EAC when constructing the example RSD program.

[1] The matching rules are those Jess rules for matching PRUs. They are different from merging rules, which are Jess rules for integrating partial programs into a complete program.

These automatically generated responsibilities of Jess facts represent the intermediate abstractions that should be created by developers when using a manual development. It is because the generated responsibilities can represent different work of different worlds. The generated responsibilities in the user-requirements world should be assumed by the target software system. The generated responsibilities in the software-design world should be assumed by programming modules, such as objects, JSP pages. The generated responsibilities in the programming-construction world should be assumed by partial programs that are created by using different technologies. The combination of these partial programs is a complete program that realizes the given business-processes responsibilities.

Figure 5-15 also shows an example of the combination that a program PA is combined from three partial programs, P1, P2, and P3. PA realizes the responsibilities of R1, R2, R3, R4, R5, and R6. Merging rules shown in Figure 5-15 serve for this purpose. However in the current version of RSDSTools, only matching rules are implemented.

5.4.2 RSD program evolution

The automation of RSD program evolution is similar to the automation of RSD program construction. While PRUs are used for finding partial programs when constructing a RSD program, they are also used for removing partial programs when evolving a RSD program. We separate the discussion into four types of evolution automation.

- **The addition of business-processes responsibility.** The addition of business-processes responsibility represents that the target system should exhibits more functions. We need to integrate new partial programs into the existing program.
- **The removal of business-processes responsibility.** The removal of business-processes responsibility represents that the target system should exhibits less functions. We need to remove the partial programs from the existing program.
- **The change of realization-development knowledge.** The change of realization-development knowledge represents that the partial programs inferred from the old development knowledge should be removed. And new

partial programs should be created from the new development knowledge and integrate into the existing program.

- **The adoption of new implementation technology.** It means that a program was originally created by one type or the combination of some types of implementation technologies, but now these implementation technologies should be replaced. The difference between the old and new implementation technologies can be two totally different platforms, such as from Java platform to .NET platform. Or they are only a minor upgrade between two versions of the same platform. In the context of RSD, this type of evolution is similar to the realization-development-knowledge evolution. It is because that in order to create a program by a different technology, we need to create another group of PRUs that describe the realization relationships and partial programs specific to the new technology. There may be some common PRUs sharing for different implementation technologies.

5.4.3 Business-processes evolution

We model business-processes evolution as the change of business-processes responsibilities. Examples of this change are various, which may include the assignment of different values to the properties of a business-processes responsibility, the modification of constraint of a business processes responsibility, a new business task adding to a business process, etc.

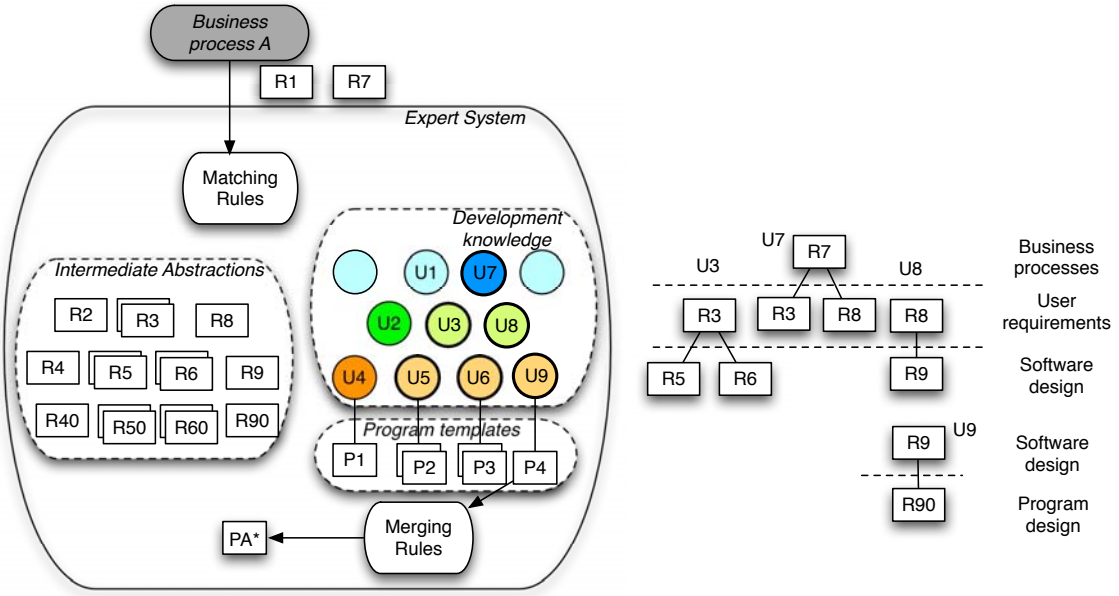
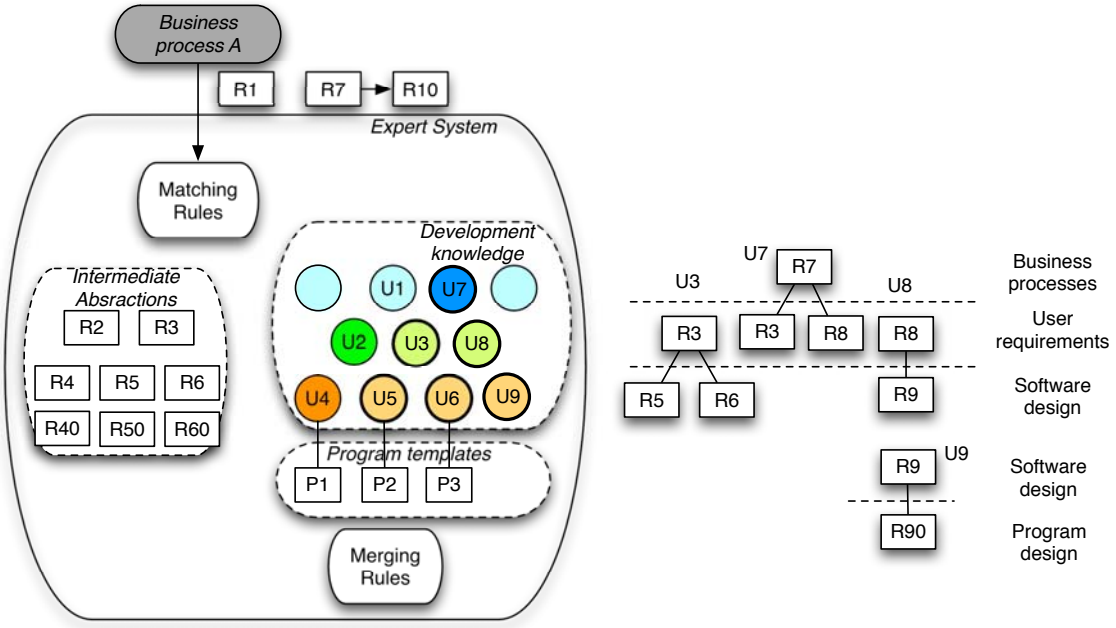
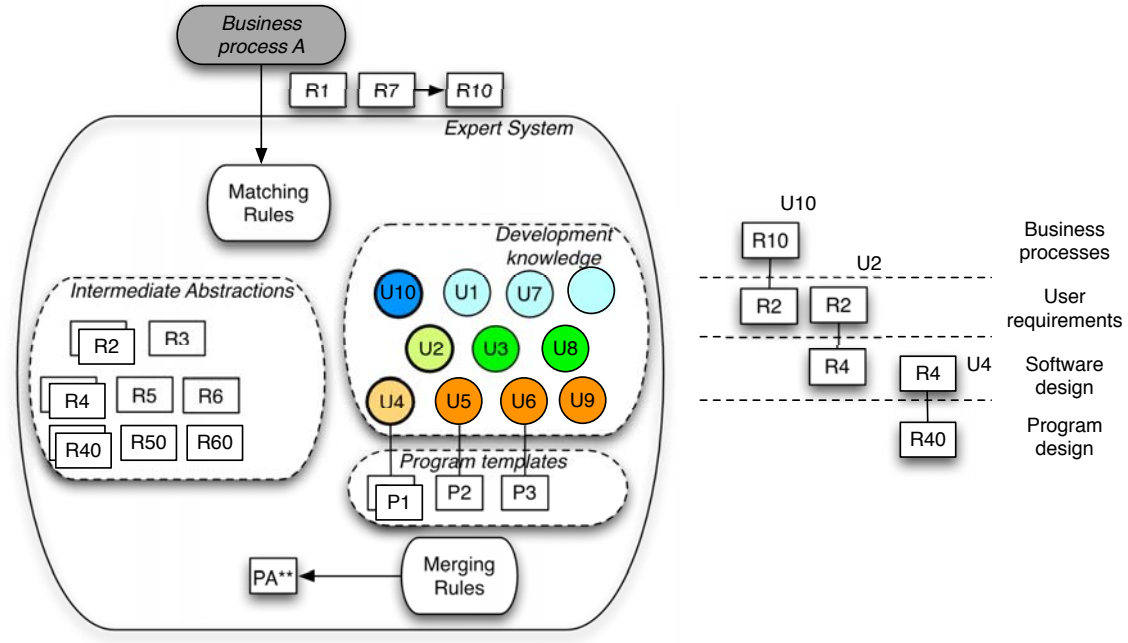


Figure 5-16. The internal work of EAC when adding new business-processes responsibility.

Figure 5-16 illustrates the internal work of the *Evolution automation component* (EAC) when adding new business-processes responsibility. It shows that there is a new business-processes responsibility R7 added, which is waiting for realization. This process is identical to the RSD program construction, which was described in Section 5.4.1. P4 is the only partial program that is new to this example. The other partial programs found, i.e. P2 and P3, are already one part of the existing program. The partial program P4 is merged with the existing program by the merging rules.


Business process A
R1
R7
R10
Expert System
Matching Rules
Development knowledge
Intermediate Absractions
R2
R3
R4
R5
R6
R40
R50
R60
U1
U7
U2
U3
U8
U4
U5
U6
U9
Program templates
P1
P2
P3
Merging Rules
U7
R7
U3
U8
R3
R3
R8
R8
R5
R6
R9
U9
R9
R90
Business processes
User requirements
Software design
Software design
Program design

(a)


Business process A
R1
R7
R10
Expert System
Matching Rules
Development knowledge
Intermediate Abstractions
R2
R3
R4
R5
R6
R40
R50
R60
U10
U1
U7
U2
U3
U8
U4
U5
U6
U9
Program templates
P1
P2
P3
PA**
Merging Rules
U10
R10
U2
R2
R2
R4
R4
U4
R40
Business processes
User requirements
Software design
Program design

(b)

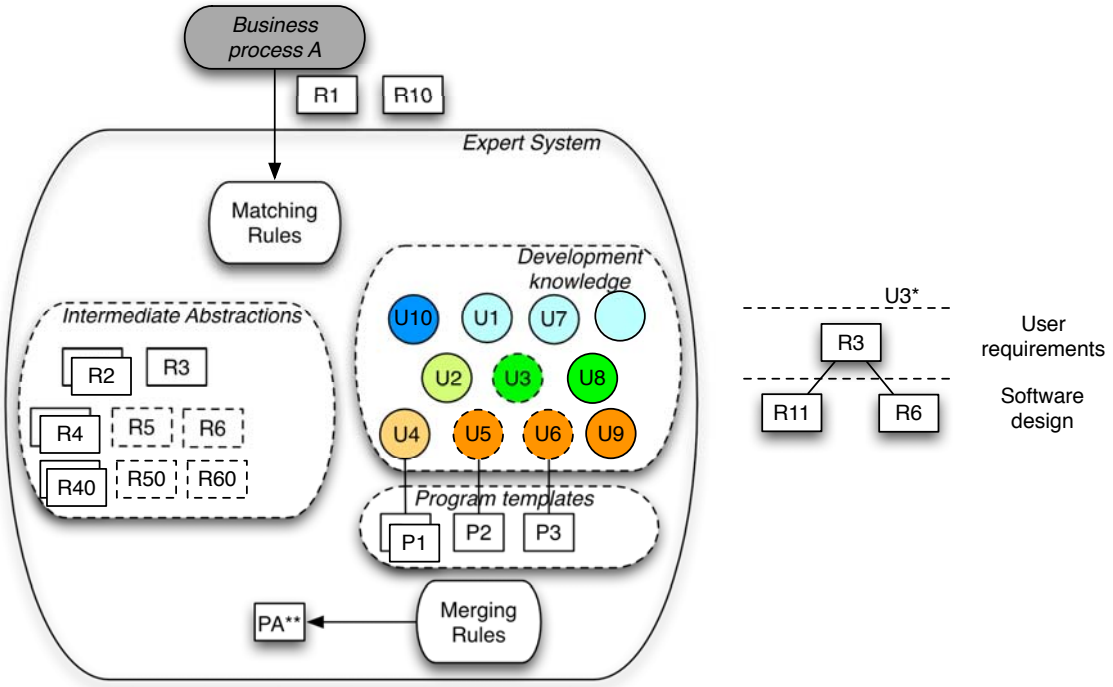
Figure 5-17. The internal work of EAC when removing business-processes responsibility.

Figure 5-17 illustrates the internal of the *Evolution automation component* (EAC) when removing existing business-processes responsibility. This example is that the business-processes responsibility R7 is replaced by a new business-processes responsibility R10. While Figure 5-17 (a) shows the removal of R7, Figure 5-17 (b) shows the additional of R10. For realizing R10, the old partial programs should be firstly located and removed. This evolution spotting activity is identical to the process described in the previous section. However, only the partial program P4 is removed. The partial programs P2 and P3 do not have to be removed because they are still used to realize R1.

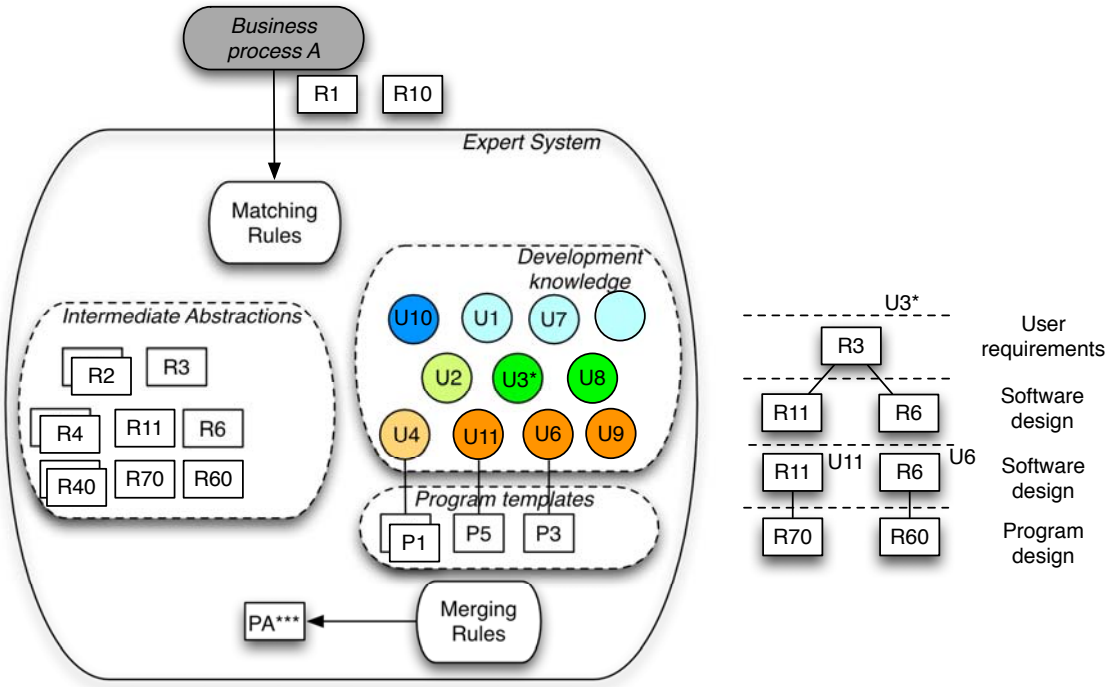
The detection of partial program preserving for other unchanged business-processes responsibilities can be easily implemented by the rule-based engine. Since RSDTools adds Jess facts of partial program representation when they are derived by PRUs, there may be multiple Jess facts asserted for one single program-construction responsibility. When there is at least one Jess fact of a program-design responsibility in Jess, the partial program of that program-design responsibility should not be removed. It can be observed by comparing between Figure 5-16 and Figure 5-17. There are multiple R50 and R60, but only one R90 in Figure 5-15. When R7 is removed in Figure 5-16, the Jess facts of R50, R60, and R90 are removed as well. However, since there is still one R50 and one R60 in Jess, their attaching partial programs P2 and P3 should not be removed.

5.4.4 Realization-development knowledge evolution

The evolution of the realization-development knowledge is represented as the changes of PRUs. The changes of PRUs may result from many reasons, such as the unsatisfied testing or execution results of the current system or the acquirement of new development knowledge during development process. This type of change is represented as the addition and the removal of PRUs.


Business process A
R1
R10
Expert System
Matching Rules
Development knowledge
U10
U1
U7
U2
U3
U8
U4
U5
U6
U9
Intermediate Abstractions
R2
R3
R4
R5
R6
R40
R50
R60
Program templates
P1
P2
P3
PA**
Merging Rules
U3*
R3
R11
R6
User requirements
Software design

(a)



(b)

Figure 5-18. The internal work of EAC when evolving the realization-development knowledge.

Figure 5-18 illustrates the internal work of the *Evolution automation component* (EAC) when evolving the realization-development knowledge. In Figure 5-18 (a), it shows that the PRU U3 is evolved to a different design. That is, R3 is newly realized by R11 and R6. The first step is evolution spotting. The partial programs that are derived from the old U3 should be located and removed. The same process illustrated in Figure 5-17 (a) is proceeded again. However, when a PRU found by Jess rules has its `old` set to TRUE, Jess rules retract (i.e remove, in Jess's term) any responsibility (R5, R6, R50, and R60 in the example) that are created from those PURs (U5 and U6 in the figure) derived after this PRU. This also removes the partial programs P2 and P3 from the complete program PA**.

In Figure 5-18 (b), it shows that after these responsibilities are removed, R3 becomes an unrealized responsibility to be realized. Jess rules will match new PRU for this

responsibility. The new U3* is matched, and R11 and R6 are instantiated. U11 and U6 are then matched to realize these two responsibilities. And finally the partial programs P5 and P3 are integrated to form the new complete program PA***.

5.4.5 Technology evolution

The automation evolution for the adoption of new implementation technology is identical to the evolution of realization-development knowledge. However, we need to add a constraint to PRUs to specify the new implementation technology used. By adding the same constraint to the unrealized business-processes responsibilities that intended to be implemented by this new implementation technology, Jess rules can match between these two. Different from MDA's idea [59, 60], RSD does not require to create a different mapping specification, one single uniformed mapping specification is used to create different variations from the same set of business processes.

5.4.6 Version control of RSD

As an important topic in configuration management [61, 62], which can be used to manage software evolution and its artifacts, we discuss version control in this section. For any model-driven approach for constructing and evolving a program, such as the one we propose in this research work, we need to shift the focus. As we can observe from some recent model-driven implementation, such as EMF in Eclipse, since code and configuration files can be automatically generated, we need to manage modeling activities in software development [63]. This may include the models and the mapping specification. In RSD, these include the humans' knowledge of PRU model and the business processes of BP model. As a supplementary topic of this research work, we only discuss the management of different versions of development artifacts in this section. We do not consider other issues, such as comparison or merge of artifacts.

To implement a version management for PRU model and BP model, we need to look back to the three evolution scenarios. When a development process only has a single evolution scenario, such as only business-processes evolution, the modeling activities merely are the addition and removal of unrealized business-process responsibilities. Since

the structure of responsibility modeling and the distinction between modeling elements are clear, all actions of addition or removal can be considered as a flatten structure, where each action can be considered as an individual version as shown in Figure 5-19. Without considering the efficiency between different concrete implementation, we can easily roll back to any version before V6 in the diagram. This also applies to realization-development knowledge realization.

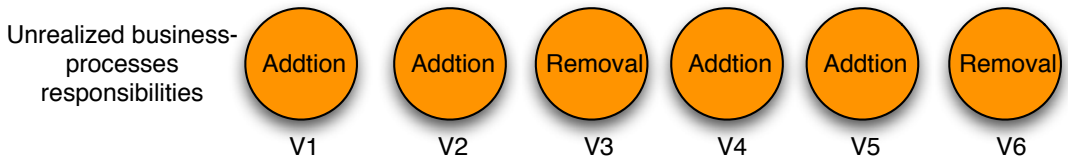


Figure 5-19. Flatten structure of single evolution scenario.

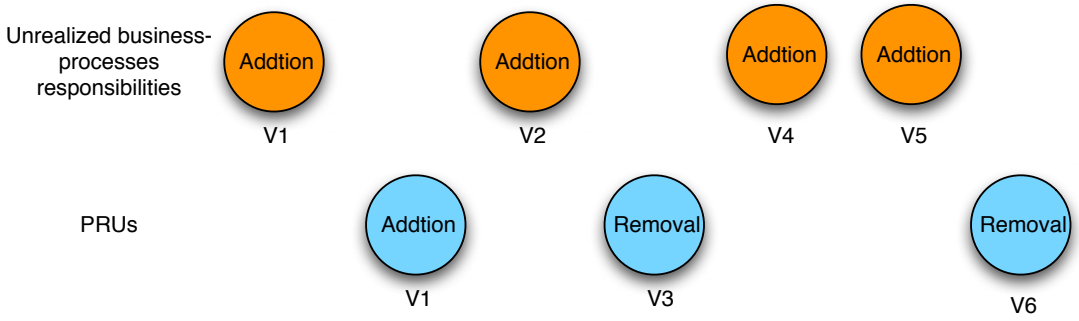


Figure 5-20. 2D structure of multiple development process.

However, when the process has different scenarios mixed in, this structure becomes a little complex. This can be observed from Figure 5-20. The complexity of the mix of multi-type evolution scenarios come from that the generated abstractions by Jess include the retract of invalid instantiated responsibilities and the re-instantiation of responsibilities. However, after a closed observation of this diagram, we know that it is actually another flatten structure, which is shown in Figure 5-21. When we see the version control of mixed evolution scenarios in a development process, we can again easily roll back a program to any previous version.

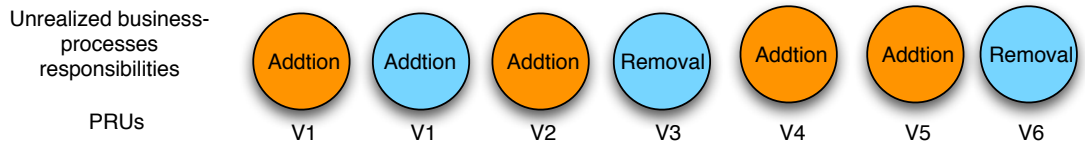


Figure 5-21. Flatten structure of mixed evolution scenarios

5.5 Summary

This chapter described the implementation the purposing approach. This implementation automates the construction/evolution of a program. We briefly described the features the implementation provides. We then described the structure of the implementation. It is constituted of three major components. Each component exhibits one part of its features. *Graphical modeling component* provides a graphical environment for modeling realization-development knowledge and business processes. *Jess code transformation component* provides the automatic transformation from the PRUs and business processes to Jess facts. *Evolution automation component* (EAC) contains Jess rules for selecting and instantiating PRUs. The input of PRUs and business processes to this implementation creates the output of all partial programs that realizes the given business processes. By this way, a program can always be kept to update to satisfy the most current business processes.

Chapter 6 Case Study

This chapter describes the evaluation of RSD. RSD is evaluated by a case study with the development of three software systems: a business-process management system (called Business-MS), a medical supporting system (called Medical-SS), and shopping-mall-on-web system (called Shopping-WS). These three systems verify the business-process evolution and realization-development knowledge evolution. Besides, the technology evolution is verified by using two different implementation technologies to create two variations from the same set of business-process responsibilities. Based on the assumption of this research work that PRUs can be reused to construct/evolve a program, we measure reusability of PRUs. In Section 6.1, we describe the basic information of the three systems. In Section 6.2, the evaluation process and the results are described. In Section 6.3, we discuss the results.

6.1 Case study overview

In this section, we describe the basic information of the three software systems.

6.1.1 Business-MS

Business-MS provides three major functional areas, sales (営業), procurement (購買), and inventory (在庫). It provides web-based UI that users can input and view business data on a web browser. The conceptual diagram of Business-MS is illustrated in Figures 6-1, 6-2, and 6-3. Business-MS is developed basing on a real-world system, which is created for a Taiwanese oil-sealed manufacture by the author. Business-MS itself has evolved to two variations, which are developed by using different implementation technologies but the same set of requirements. The first variation is a JavaServer Pages (JSP) with JavaBeans implementation. The second variation is a JBoss Seam implementation.

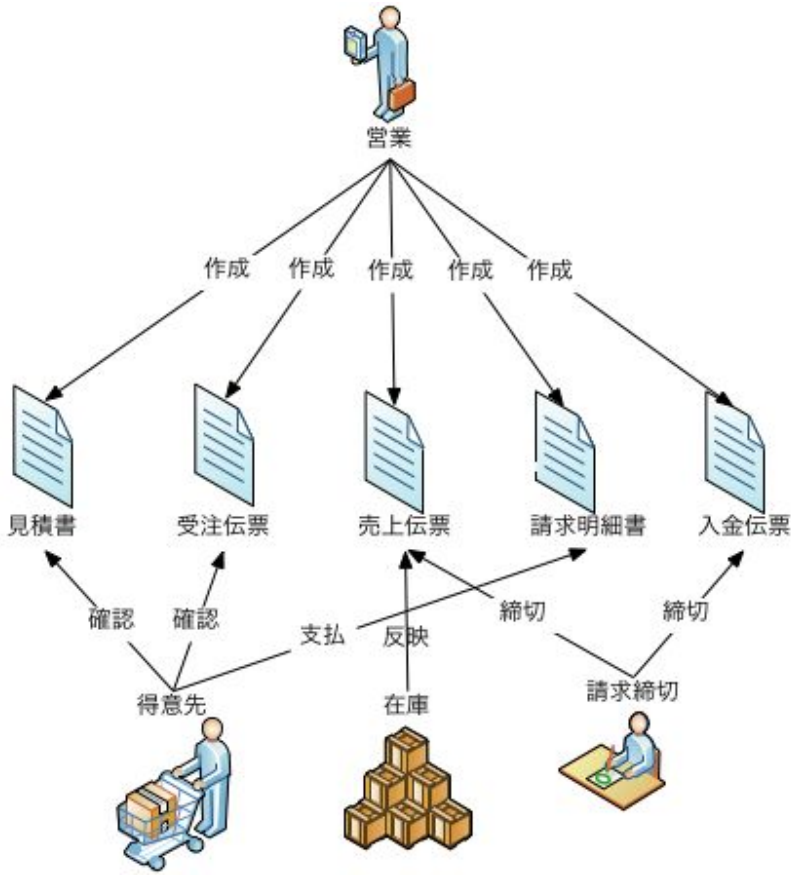


Figure 6-1. Conceptual flow of Business-MS for Sales document processing

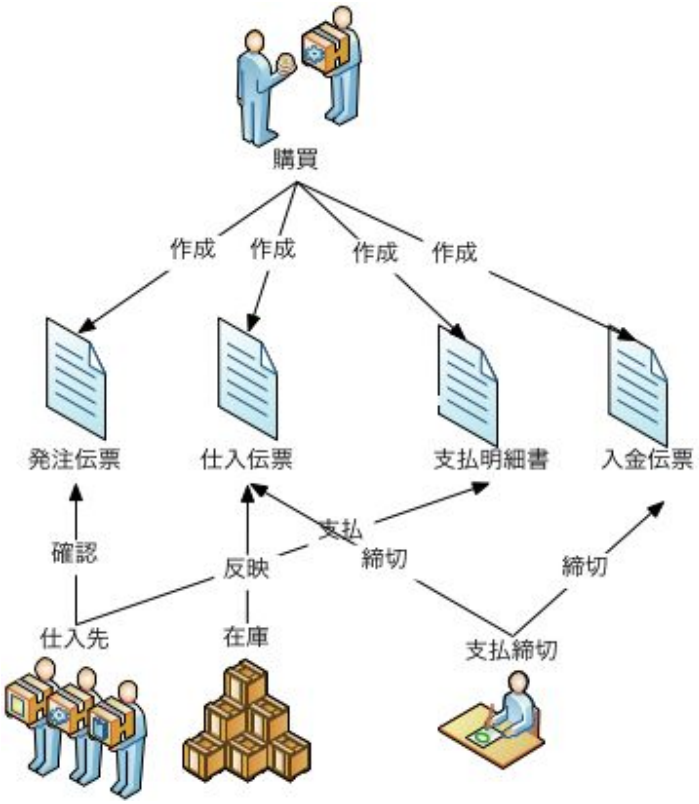


Figure 6-2. Conceptual flow of Business-MS for procurement document processing

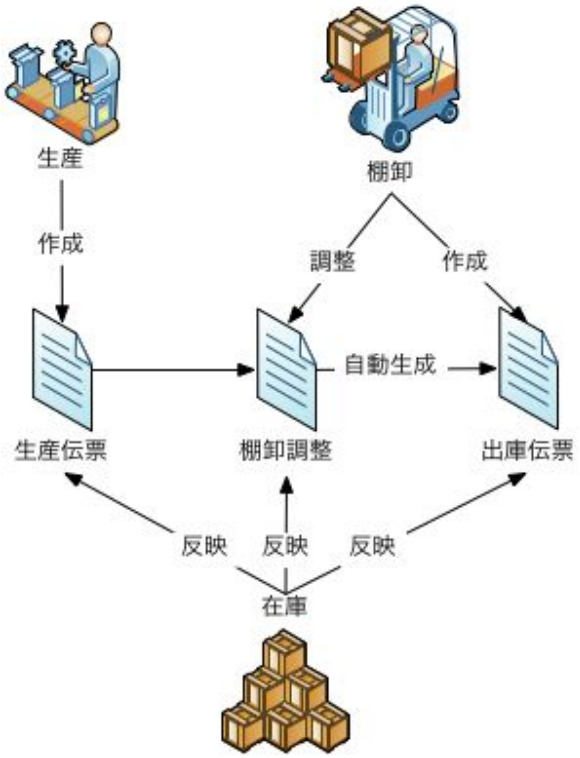


Figure 6-3. Conceptual flow of Business-MS for inventory document processing

For Business-MS, we have created several documents. We list each document and the overview of their contents in Table 6-1. In order to maintain the layout of each document, they are not included in this dissertation, but providing as separate volumes.

Table 6-1. Document list of Business-MS. All of these three volumes are provided as companion documents.

Document	Overview
Vision and scope	It is used to communicate the high-level vision with the stakeholders of Business-MS. It also confines the scope of features that will be appeared in Business-MS
Use cases	It contains all of the use case descriptions of BMS.
Design book	It describes the design process of Business-MS. It is especially created for the second

Document	Overview
	version of Business-MS. It also describes some examples of detailed design in UML diagrams.

6.1.2 Medical-SS

Medical-SS is also a web-based business supporting application. Medical-SS is developed basing on a research project [6]. In this research project, Lori A. C. et al. use a process-modeling language, called Little-JIL, for describing the collaboration among different actors that are involving in the blood-transfusion process. Based on this research project, we create an imaginary project for processing information is generated in the process. Medical-SS also provides web-based UI that users can input and view medical data on a web browser. The conceptual diagram of Medical-SS is illustrated in Figure 6-4. Similar to Business-MS, Medical-SS has evolved to two variations, which are developed by using two different implementation technologies but the same set of requirements. The first variation is a JavaServer Pages (JSP) with JavaBeans implementation. The second variation is a JBoss Seam implementation.

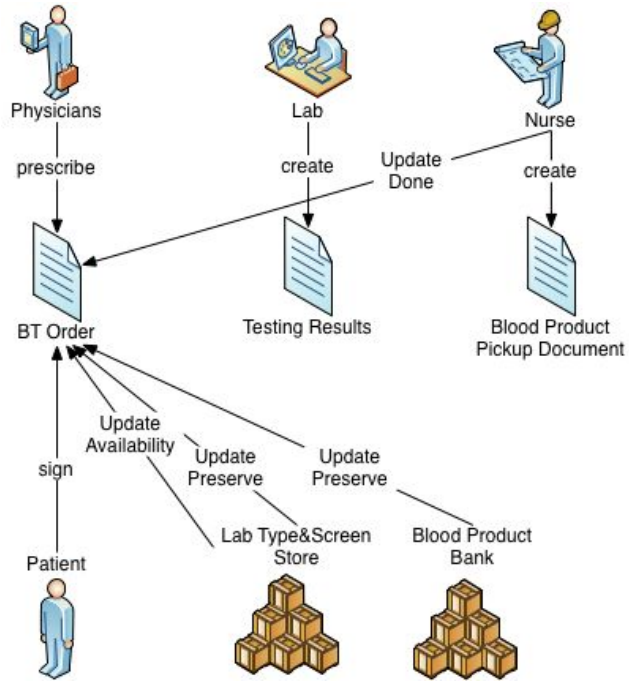


Figure 6-4. Conceptual flow of Medical-SS.

6.1.3 Shopping-WS

Shopping-WS is a web-based shopping mall for people to open customizable on-line stores. Each shop opens on Shopping-WS is impendent from other shops, however, they have same features provided by Shopping-WS. Shopping-WS helps shop owners manage their item catalog, orders, and customer records. Identical to two other systems, it provides web-based UI that users can input and view business data on a web browser. The conceptual diagram of Shopping-WS is illustrated in Figure 6-5. Shopping-WS is also considered to be deployed commercially in future. Different from the previous two systems, Shopping-WS has one only implementation by using JBoss Seam.

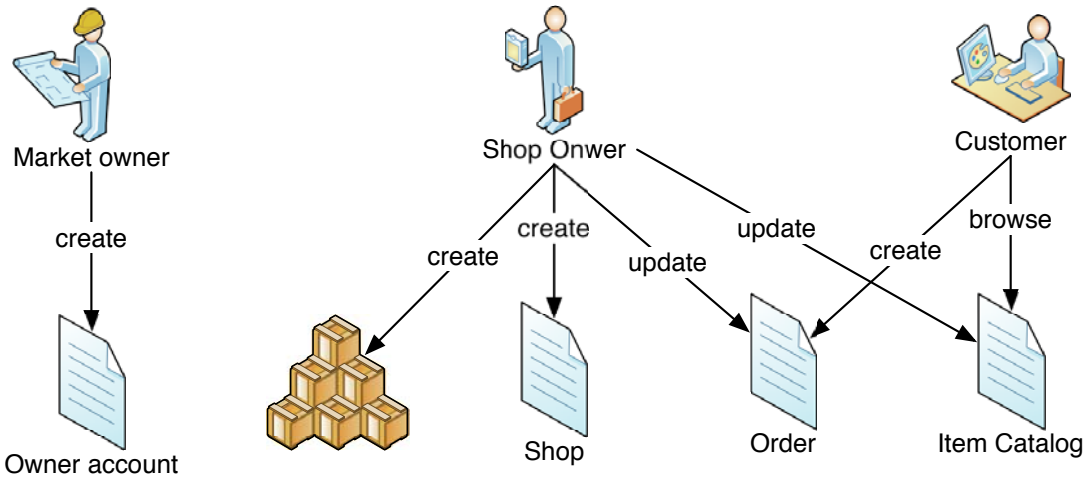


Figure 6-5. Conceptual flow of Shopping-WS.

6.2 Evaluation

The application process of RSD is as follows. For all three systems, we input business-process responsibilities into Jess. Jess infers the partial programs from the three categories of PRUs. This business-process evolution has been illustrated in Figure 5-16 and Figure 5-17 before. When an unrealized responsibility, which may belong to any of three worlds: business processes, user requirements, and software design, cannot be realized by any current PRU, we create a new PRU for this unrealized responsibility. When one part of the design or implementation of the target system does not generate satisfactory results, we remove the PRU that this part and create a new PRU. This development knowledge evolution has been illustrated in Figure 5-18. We realized the business processes sequentially. That is, S-1 is realized first, and then S-2, and so forth. When creating PRUs, we always create a base PUR first because it provides a most-general-case for reusing. A variation PRU is created only when its base cannot generate satisfactory results. When knowledge is evolved, that is, an existing PRU is removed or a new PRU is added, RSDTools automatically apply the new set of PRUs to any business-processes that has been realized.

The purpose of this research work is to improve software evolution by managing the complex relationships between abstractions of different stages. To this end, we proposed and implemented software-evolution automation for high-level abstractions realization in a program. The proposing approach reuses humans' knowledge stored as PRUs for constructing/evolving a program. Therefore, by measuring the quantity of the case study that is related to reusability of development artifacts, we can understand the effectiveness of it. We list the number of required PRUs, i.e. PRUs that are necessary to construct a program for realizing responsibilities of one business process, and new PRUs, i.e. PRUs that are newly created in that business process of each system from Tables 6-2 to 5-6 A high reused ratio represent that there more PRUs reused than PRUs created for realizing a business process.

Table 6-2. Numbers of required and new PRUs for each business processes for the JSP system of Business-MS

Business process	Number of Business-processes responsibilitie	Required business-processes PRUss	New business-processes PRUss	Reused ratio of business-processes PRUss	Required user-requirements PRUss	New user-requirements PRUss	Reused ratio of user-requirements PRUss	Required software-design PRUss	New software-design PRUss	Reused ratio of software-design PRUss
Create order	5	5	5	0%	8	8	0%	16	12	25%
Modify order	5	5	3	40%	7	3	57%	11	2	82%
Cancel order	5	5	1	80%	6	1	83%	11	0	100%
Request catalog	3	3	0	100%	4	0	100%	6	0	100%
Review orders	3	3	0	100%	3	0	100%	6	0	100%
Create quotation	5	5	0	100%	8	0	100%	16	0	100%
Modify quotation	5	5	0	100%	7	0	100%	11	0	100%
Create sales order from quotation	3	3	0	100%	4	0	100%	8	3	63%
Create invoice from sales order	3	3	0	100%	4	0	100%	8	0	100%
Create return document form sales order	3	3	0	100%	4	0	100%	8	0	100%
Manage customer accounts	7	7	2	71%	10	2	80%	18	1	94%
Manage product catalog	7	7	0	100%	10	0	100%	18	0	100%
Create purchase order	5	5	0	100%	8	0	100%	16	0	100%
Modify purchase order	5	5	0	100%	7	0	100%	11	0	100%
Cancel purchase order	5	5	0	100%	6	0	100%	11	0	100%
Create invoice/shippment receipt	3	3	0	100%	8	0	100%	8	0	100%
Manage suppliers	7	7	0	100%	10	0	100%	18	0	100%
Create inventory item	5	5	0	100%	8	0	100%	16	0	100%
Adjust stocking quantity of inventory items	4	4	0	100%	6	0	100%	11	0	100%
Transfer inventory items	4	4	0	100%	6	0	100%	11	0	100%
Create shipping document from sales order	3	3	0	100%	4	0	100%	6	0	100%
Log in	3	3	3	0%	4	2	50%	6	3	50%

Table 6-3. Numbers of required and new PRUs for each business processes for the JSP system of Medical-SS

Business process	Number of Business Processes	Required business-processes PRUss	New business-processes PRUss	Reused ratio of business-processes PRUss	Required user-requirements PRUss	New user-requirements PRUss	Reused ratio of user-requirements PRUss	Required software-design PRUss	New software-design PRUss	Reused ratio of software-design PRUss
Create BT order	4	4	0	100%	5	1	80%	10	0	100%
Modify BT Order	5	5	0	100%	7	0	100%	11	0	100%
Cancel BT order	5	5	0	100%	6	0	100%	11	0	100%
Create blood pickup document from BT order	4	4	0	100%	5	0	100%	10	0	100%
Sign BT order consensus	5	5	1	80%	6	1	83%	10	1	90%
Update testing results	5	5	0	100%	4	0	100%	11	0	100%
Log in	3	3	0	100%	4	0	100%	6	0	100%

Table 6-4. Numbers of required and new PRUs for each business processes for the JBoss Seam system of Shopping-WS

Business process	Number of Business Processes	Required business-processes PRU	New business-processes PRU	Reusable percentage of business-processes	Required user-requirements PRU	New user-requirements PRU	Reusable percentage of user-requirements	Required software-design PRUss	New software-design PRUss	Reused ratio of software-design PRUss
Create shop	8	8	4	50%	15	6	60%	33	33	0%
Modify shop	8	8	0	100%	15	0	100%	33	1	97%
Create sales order	5	3	0	100%	8	0	100%	16	0	100%
Modify sales order	5	4	0	100%	7	0	100%	11	1	91%
Create invoice from sales order	3	4	0	100%	4	4	0%	8	0	100%
Create return document form sales order	3	4	0	100%	4	0	100%	8	0	100%
Manage customer accounts	7	6	0	100%	10	0	100%	18	0	100%
Create product item	5	3	0	100%	8	0	100%	16	0	100%
Manage product item	7	5	0	100%	7	0	100%	11	0	100%
Adjust stocking quantity of product items	4	3	0	100%	6	0	100%	11	0	100%
Create shipping document from sales order	3	4	0	100%	4	0	100%	6	0	100%
Create purchase order	5	3	0	100%	8	0	100%	16	0	100%
Modify purchase order	5	4	0	100%	7	0	100%	11	0	100%
Cancel purchase order	5	4	0	100%	6	0	100%	8	0	100%
Log in	3	3	0	100%	4	0	100%	6	3	50%

Table 6-5. Numbers of required and new PRUs for each business processes for the JBoss Seam system of Business-MS

Business process	Number of Business Processes	Required business-processes PRUss	New business-processes PRUss	Reused ratio of business-processes PRUss	Required user-requirements PRUss	New user-requirements PRUss	Reused ratio of user-requirements PRUss	Required software-design PRUss	New software-design PRUss	Reused ratio of software-design PRUss
Create order	3	3	0	100%	5	5	0%	9	7	22%
Modify order	4	4	0	100%	6	2	67%	9	3	67%
Cancel order	4	4	0	100%	6	0	100%	9	0	100%
Request catalog	2	3	0	100%	4	0	100%	5	0	100%
Review orders	4	4	0	100%	5	0	100%	8	0	100%
Create quotation	3	3	0	100%	4	0	100%	7	0	100%
Modify quotation	4	4	0	100%	5	0	100%	9	0	100%
Create sales order from quotation	4	4	0	100%	5	2	60%	9	3	67%
Create invoice from sales order	4	4	0	100%	5	0	100%	9	0	100%
Create return document form sales order	4	4	0	100%	5	0	100%	9	0	100%
Manage customer accounts	5	5	0	100%	6	3	50%	10	4	60%
Manage product catalog	5	5	0	100%	6	0	100%	10	0	100%
Create purchase order	3	3	0	100%	4	0	100%	8	0	100%
Modify purchase order	4	4	0	100%	6	0	100%	8	0	100%
Cancel purchase order	4	4	0	100%	6	0	100%	8	0	100%
Create invoice/shippment receipt	4	4	0	100%	6	0	100%	8	0	100%
Manage suppliers	5	5	0	100%	6	0	100%	10	0	100%
Create inventory item	3	3	0	100%	4	0	100%	8	0	100%
Adjust stocking quantity of inventory items	3	3	0	100%	4	2	50%	8	3	63%
Transfer inventory items	3	3	0	100%	4	2	50%	8	3	63%
Create shipping document from sales order	3	3	0	100%	4	0	100%	8	0	100%
Log in	2	2	0	100%	3	3	0%	5	5	0%

Table 6-6. Numbers of required and new PRUs for each business processes for the JBoss Seam system of Medical-SS

Business process	Number of Business Processes	Required business-processes PRUss	New business-processes PRUss	Reused ratio of business-processes PRUss	Required user-requirements PRUss	New user-requirements PRUss	Reused ratio of user-requirements PRUss	Required software-design PRUss	New software-design PRUss	Reused ratio of software-design PRUss
Create BT order	4	4	0	100%	5	0	100%	9	0	100%
Modify BT Order	5	5	0	100%	7	0	100%	9	0	100%
Cancel BT order	5	5	0	100%	6	0	100%	9	0	100%
Create blood pickup document from BT order	4	4	0	100%	5	0	100%	9	0	100%
Sign BT order consensus	5	5	0	100%	6	0	100%	10	0	100%
Update testing results	5	5	0	100%	4	0	100%	7	0	100%
Log in	3	3	0	100%	4	0	100%	5	0	100%

6.3 Discussion

We list what we observed from the evaluation in this section.

To maximize reusability, it is important to firstly implement those architecturally significant business processes. It is because that these business processes can reveal more responsibilities, therefore more PRUs, for reusing in other business processes. For all three systems (including the variations), we always implemented the architecturally significant business processes first. They generated most PRUs that can be reused for other business processes. It can be observed from Figures 6-6 to 6-11. These figures are drawn from Table 6-2 to 6-6. They showed that when there is a business process that has different business task than others, there are always more new PRUs created. For example, the JSP system of BPMS and the Seam system of WMS have the lowest reusable ratio than other business processes. More specifically, in Figure 6-7 and Figure 6-11, the reused ratios of the first two or three business processes are lower than other business processes.

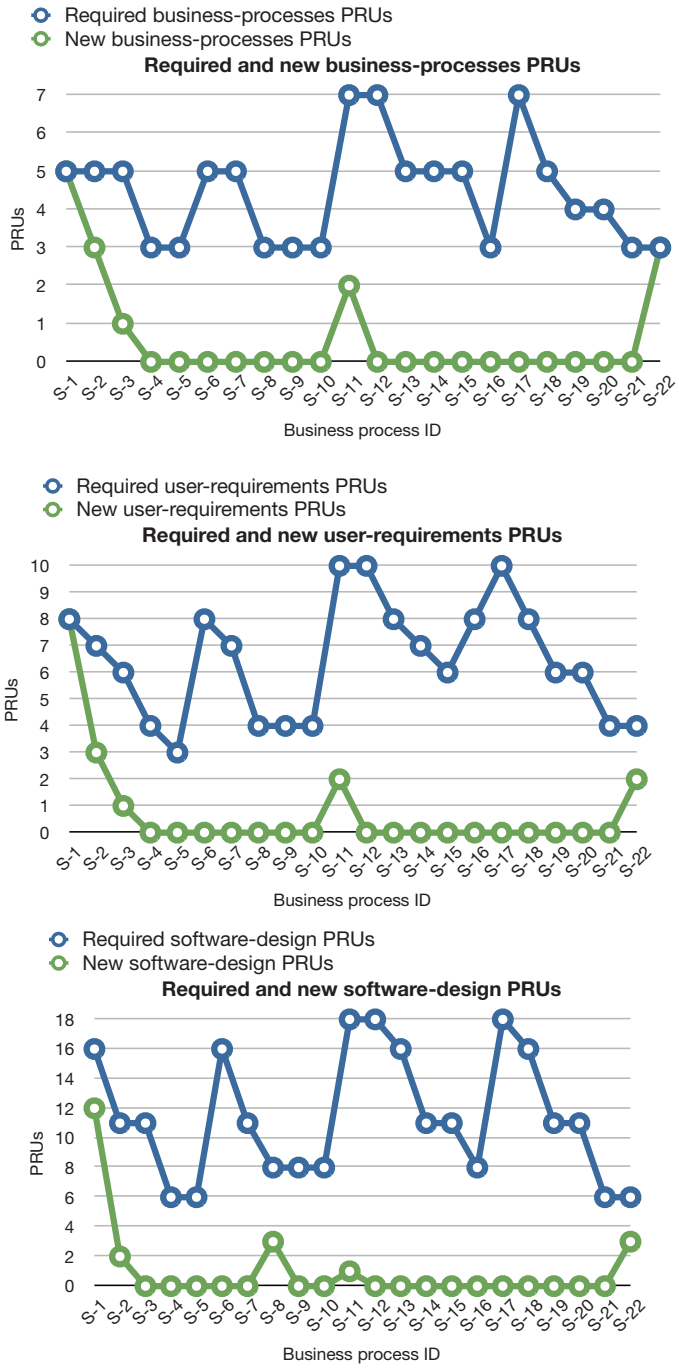


Figure 6-6. Required and new PRUs of the JSP system of Business-MS.

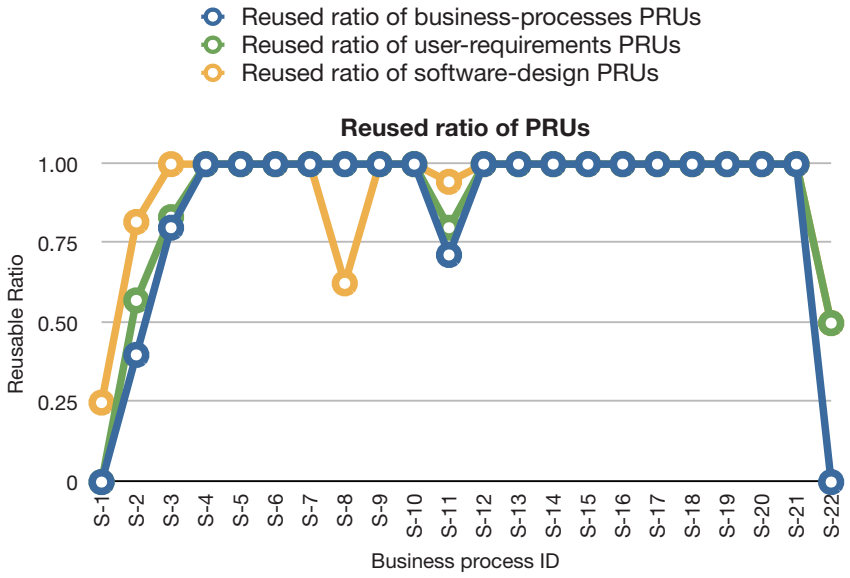


Figure 6-7. Reused ratios of the JSP system of Business-MS.

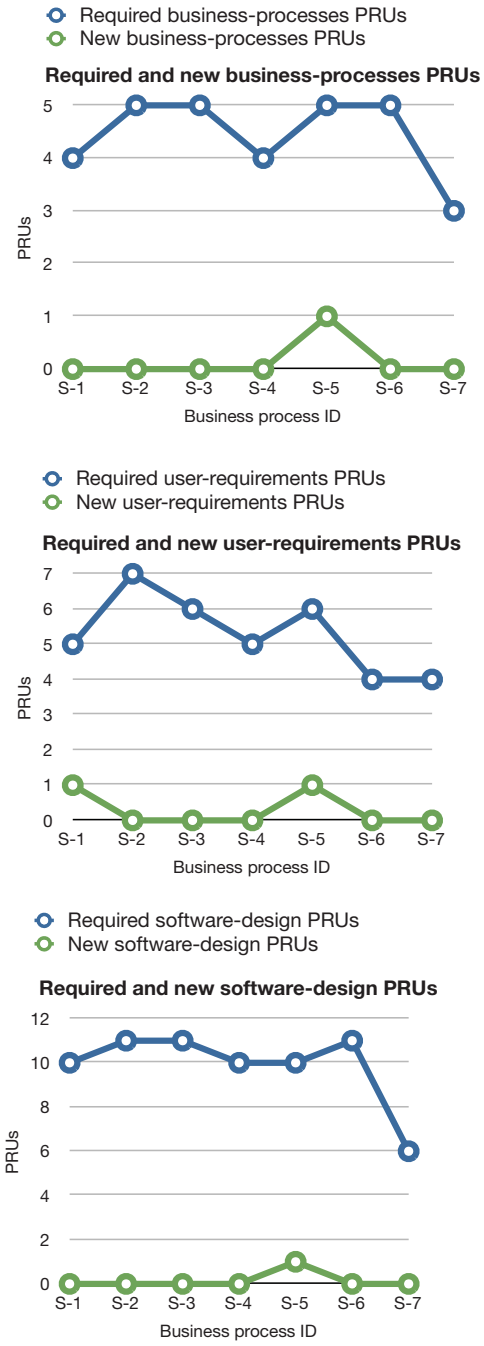


Figure 6-8. Required and new PRUs of the JSP system of Medical-SS.

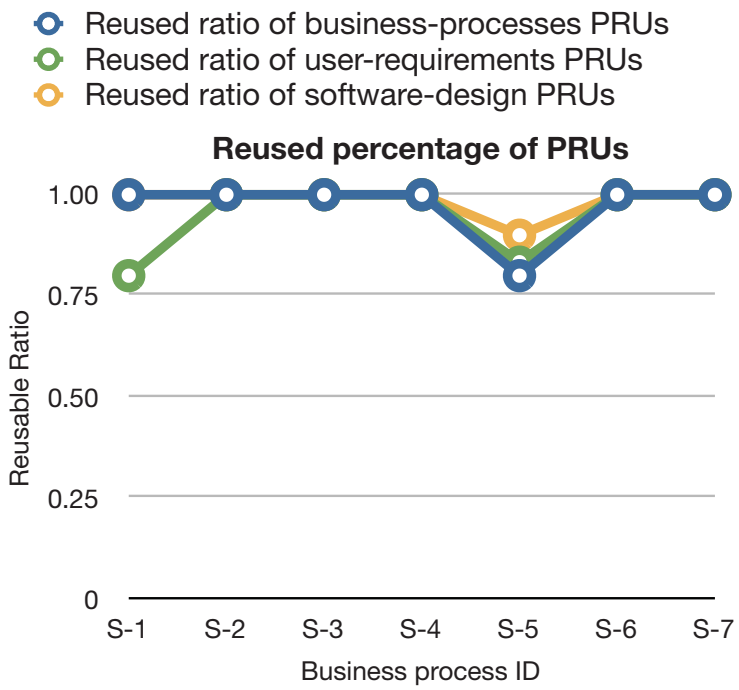


Figure 6-9. Reused ratios of the JSP system of Medical-MS.

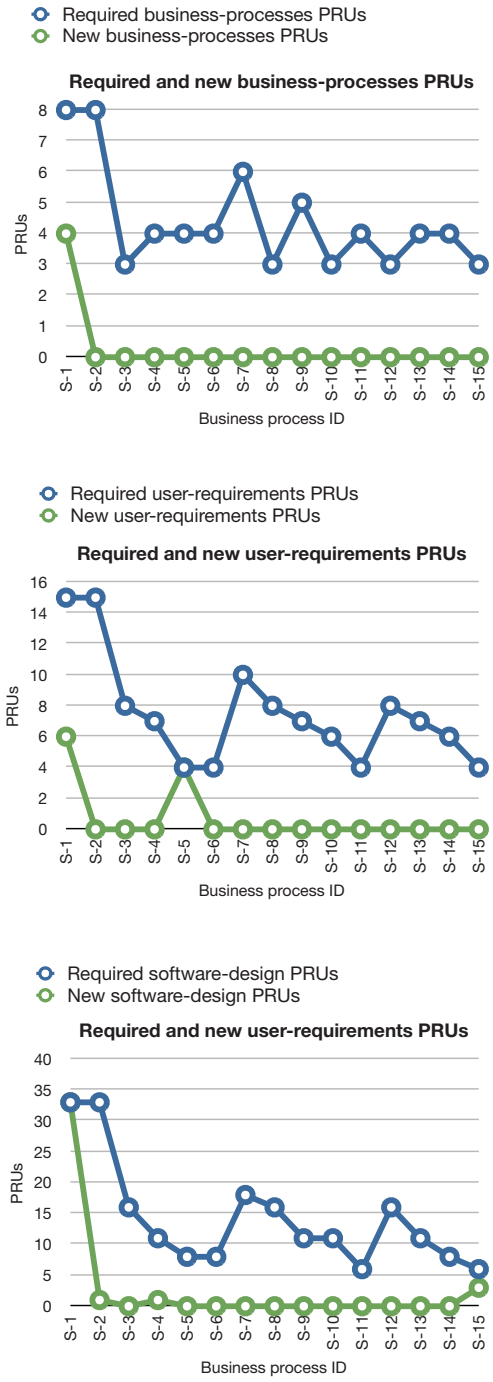


Figure 6-10. Required and new PRUs of the JBoss Seam system of Shopping-WS.

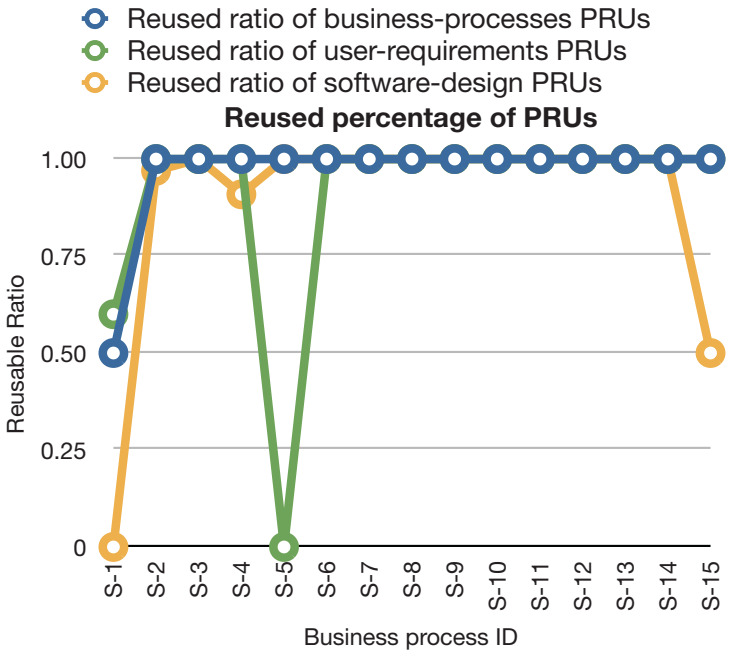


Figure 6-11. Reused ratios of the JBoss Seam system of Business-MS.

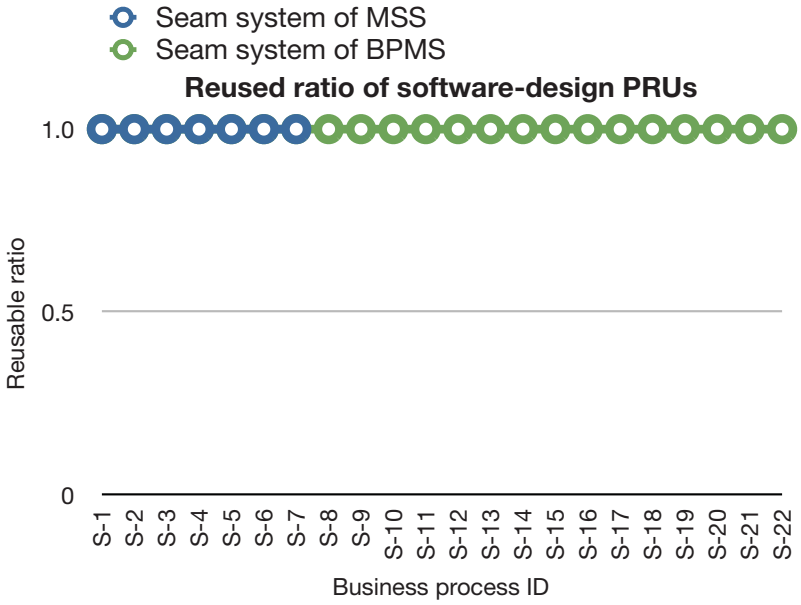


Figure 6-12. Reused ratios of the JBoss Seam systems of both Business-MS and Medical-SS.

By observing Tables 6-2 to 6-6, we know that both business-processes and user-requirements PRUs can be reused for developing systems implemented on different technologies. It is because responsibility abstracts away the implementation-technology differences. More specifically, responsibilities only define what should be accomplished by different types of entities. They did not specify how they should be accomplished. How they are accomplished can be lately decided by those software-design PRUs. These PRUs have partial programs implemented on different technologies.

Although it cannot be directly observed how rule engine simplifies the work of knowledge application, it is true that for those business processes with high reusable ratios they are almost fully-automated. Rule engine automatically infers the implementation of the given business-processes responsibilities.

All in all, the evaluating results show that PRUs are an effective tool for developing business systems. Responsibility simply represents the abstractions of different worlds. PRUs capture the realization development knowledge. There is no necessary to create "traditional" modeling diagrams, such as UML class diagrams or UML sequential diagrams. The results also show that PRUs are also an effective tool for reducing the repetition in the development process. Once a PRU is created for one part of a system, it can be reused for developing other parts of the same system or even a different system.

6.4 Summary

This chapter described the case study that evaluated the effectiveness of the proposing approach. In this research work, we made a claim that by using RSD in the three evolution scenarios, the development of a business software system can be dynamically satisfied by the collection of realization-development knowledge. In the case study, RSDTools, which is the implementation based on the proposing approach, is used for the development of the three software systems in the case study. The evaluation results showed that PRU is not only capable of constructing a program, but also be capable of evolving a program by reducing the repetition in realization development.

Chapter 7 Summary and Future Work

The purpose of this research work is to automate software evolution by managing the complex relationships between abstractions of different stages. To this end, we proposed and implemented software-evolution automation for high-level abstractions realization in a program. In this research work, we used a case study of three software systems to verify the claim we made, i.e. by using the purposing approach, the development of a business system can be dynamically satisfied by using the collection of realization-development knowledge under the three evolution scenarios. The evaluation results showed that developers can freely modified new development knowledge to realize any unrealized business responsibility without invalidating current realization under the three evolution scenarios mentioned above.

The evaluation results also showed that the combination of the three fundamental theories proposing in this research work can be effectively used to fill the gap in software development, i.e. the necessary to eliminate the repetition of creating similar realization relations between high-level and low-level abstraction many times. These three fundamental theories are

- To eliminate the gap between different types of abstractions, we proposed responsibility modeling, a modeling approach based on a single-type paradigm, responsibilities.
- To simplify the evolution of a program, we proposed capturing the connections of the high-level abstraction realization and implementation artifacts as reusable and composable knowledge by the paradigm of responsibility.
- To automate the reusing and composing of knowledge, we proposed using rule engine for encoding the realization-development knowledge.

The construction steps of the proposing approach are

1. To design the basic framework.
2. To implement the tool for supporting the automated construction/evolution of a program.
3. To develop a case study with three systems

7.1 To design the basic framework

To help developers to capture realization-development knowledge they acquire in the development process, a modeling language for capturing the realization-development knowledge is created. It also includes the definition of four connected worlds, where each world belongs to a distinct context that is important to the constructing of a program for the business domain. A set of graphical notations for visually modeled knowledge is also created. Finally, it defines the idea of using many small pieces of realization-development knowledge for constructing/evolving a program.

In Chapter 3, we described this modeling language. The modeling language provides the modeling constructs for capturing responsibility that should be performed by entities and for capturing various types of relationships among entities, which include collaboration, realization, and constraints. This modeling language is supplemented with a set of graphical notations for visually modeling. One important idea of RSD is the modularization of humans' knowledge about realization-development between different worlds (i.e. different stages of software development process). We create a construct, called parameterized realization unit (PRU for short) for this purpose. We explained that the development process when using RSD can be separated into two activates: the domain modeling and the application modeling. In the domain modeling, developers create reusable PRUs. In the application modeling, developers reuse PRUs for constructing/evolving a program.

In Chapter 4, we detailed the usage of PRUs in program construction/evolution. We showed that when a program is constructed by only using PRUs, it contains all the necessary information for managing the complex relationships among abstractions of

different worlds. Therefore, it is also easier to be evolved than by using other traditional approaches. This chapter also described what a *parameterized* PRU is. A *parameterized* PRU has one or more properties set to no value when it is created, and is instantiated when it is used. It is used to create customizable realization relationship. It simplifies the construction/evolution of a program.

7.2 To implement the tool for supporting the automated construction/evolution of a program

The implementation of the purposing approach is a tool for automating program construction/evolution. Based on the third theory, RSDTools is implemented on a rule engine for reasoning a program that realizes the given business processes. Two types of knowledge are encoded in this rule-based engine. The first type is PURs, which represent the realization development knowledge. The second type is the matching scheme for selecting and instantiating a PRU. By the combination of these two, this tool achieves the research goal of realization-evolution automation.

In Chapter 5, we described the features this implementation provides, the structure it is constructed, and the internal work of the implementation for automating the three evolution scenarios. The implementation is constituted of three components. *Graphical modeling component* provides a graphical environment for modeling realization-development knowledge and business processes. *Jess code transformation component* provides the automatic transformation from the PRUs and business processes to Jess facts. *Evolution automation component* (EAC) contains Jess rules for selecting and instantiating PRUs. The input of PRUs and business processes to this implementation creates the output of a program that realizes the given business processes. By this way, a program always satisfies the most updated business processes.

7.3 To develop a case study with three systems

In this research work, we made a claim that by using RSD, in the three evolution scenarios, the development of a business software system can be dynamically satisfied by the collection of realization-development knowledge. That is, by using RSD, developers can freely modify new development knowledge to realize any unrealized business responsibility without invalidating current realization under the scope of the proposing approach. Therefore, the verification of this claim becomes the evaluation of the effectiveness of RSD.

In Chapter 6, we described the basic information of the three systems. In the case study, the three systems are all web-based systems but for different purposes. Business-MS is a business-process management system. Medical-SS is a medical supporting system Shopping-WS is shopping-mall-on-web system. We used diagrams to show the conceptual workflow of them. The three systems verified this claim in the business-process evolution and the realization-development knowledge evolution. Besides, the technology evolution is verified by using two different implementation technologies, i.e. JSP and JBoss Seam, to create two variations from the same set of business-process responsibilities.

In the evaluation, we measured reusability of RSD by counting the number of required PRUs and newly created PRUs in each business process of each software system. We also calculated the reused ratios, which are the proportion of the newly created PRUs in each business process. The results showed a program can be constructed by reusing realization-development knowledge. From the high reused ratios in the results, we know that PRU is not only capable of constructing a program, but also be capable of evolving a program by reducing the repetition in realization development.

7.4 Contribution

The proposing approach shows its significance by the following points:

- A single-type paradigm is capable for software modeling, although software development is multi-context in nature. This research work showed that responsibility,

which is already a proven concept in software object design, can be used to bridge the gap between different worlds (i.e. different stages in software development process). This provides support for the first fundamental theory of the proposing approach.

- A program can be constructed by using the modularization of realization-development knowledge. The using of PRUs as the units in software development is capable for both constructing/evolving a program. PRUs also reduce the repetition in abstraction-realization development. This provides support for the second fundamental theory of the proposing approach.
- Rule-based engine can effectively infer humans' knowledge for software development. With a few of pre-loaded rules by our implementation, rule-based engine automatically infer partial programs for realizing given business processes. A program can always be kept to update to satisfy the ever-changing environment under the scope of this research. This provides support for the third fundamental theory of the proposing approach.

The combination of these points is a development approach for automating program evolution for the business domain.

7.5 Future Work

This research work provides a new way for modeling complex relationships among abstractions and for constructing/evolving a program. However, the scope of this work is limited. We only show its applicability in the business domain. We also only showed its applicability for modeling functional requirements, design, and implementation. We need to consider the modeling of non-functional requirements by responsibility modeling. Besides, since responsibility is a common and easy-to-comprehend concept, it is expected to apply and expand responsibility modeling to model humans' social behavior and system interactions, for example, the using of PRUs for the regulation-compliance implementation. It is also expected to apply the basic concepts of RSD to other domains, such as embedded system. As we did in this research work, the relationships among a hardware system and its

components, the users of this hardware system, the software system of this hardware system are all possible to be modeled as responsibilities.

This current research work mainly focused on high-level modeling. Therefore, one missing puzzle is about partial program integration. We only showed the selection of partial programs for realizing given business-processes responsibilities. Therefore, another direction is to directly use responsibility in software implementation. More specifically, we can use responsibility to model finer-grained programming constructs, such as looping, conditional statements etc. These responsibilities can be considered as the higher-level representation of the basic programming constructs. This higher-level representation provides a neutral media for different programming languages. By this neutral media, developers have no necessary to learn the semantic details or syntactical differences of programming languages, but only by reusing these responsibilities to compose a program that can satisfy responsibilities of other worlds. This highly integrated development paradigm should provide a better platform for software evolution.

Another possible extension of this research work is the combination of ontology. Ontology is the tool for representing domain-specific knowledge which can be used in artificial intelligence. Its purpose is to share understanding of concepts that are important to some domains [64]. This purpose shows the validity to move our PRU definition (the metamodel described in Chapter 3) to an ontology-based definition. The direction of this approach can be the combination of the modeling of each world (business, system, design, and construction) where the combination will be the result of the collection of PRUs. The benefit of this approach is that each world is defined by using a modeling language that is suitable to one specific domain. However, it may also defeat our purpose to use one-single paradigm for modeling development knowledge.

References

[1] M. Lehman and J. C. Fernandez-Ramil, Software Evolution, "Software Evolution and Feedback: Theory and Practice", Wiley, 2006

[2] COM: Component Object Model Technologies, Microsoft. http://www.microsoft.com/com/default.mspx

[3] Enterprise JavaBeans Technology, Sun Microsystems. http://java.sun.com/products/ejb/

[4] R. Wirfs-Brock and A. McKean, "Object-Oriented Design: A Responsibility-Driven Approach", OOPSLA 1989.

[5] R. Wirfs-Brock and A. McKean, Object Design : Roles, Responsibilities, and Collabs, Addison-Wesley, Boston, 2003.

[6] Lori A. C. et al. "Process Programming to Support Medical Safety: A Case Study on Blood Transfusion", Proceedings of the Software Process Workshop 2005.

[7] JavaServer Pages Technology, Sun Microsystems. http://java.sun.com/products/jsp/

[8] JavaBeans, Sun Microsystems. http://java.sun.com/products/javabeans/

[9] Hans Bergsten, JavaServer Pages, Third Edition, Sebastopol, Calif. : O'Reilly, c2004.

[10] JBoss Seam, Red Hat Middleware, LLC., http://www.jboss.com/products/seam

[11] Michael Juntao Yuan (Author), Thomas Heute (Author), JBoss(R) Seam: Simplicity and Power Beyond Java(TM) EE, Prentice Hall PTR; 1 edition (April 26, 2007)

[12] M. Lehman and J. C. Fernandez-Ramil, Software Evolution and Software Evolution Processes, Annals of Software Engineering, Volume 14, Numbers 1-4 / December, 2002

[13] M.M. Lehman, the Programming Process, IBM Research Report RC2722, 1969, IBM Research Center, NY.

[14] M.M. Lehman, Program Life Cycle and Laws of Software Evolution, Proc. IEEE Spec Iss. On Software Eng., Vol 68, no 9, Sept. 1980, pp. 1060-1076.

[15] M.M. Lehman, Program Evolution and its Imact on Software Engineering, Proc. ICSE 1976.

[16] M.M. Lehman and J.F. Ramil, An Approach to a Theory of Software Evolutution

[17] C.K.S. Chong Hok Yuen, Phenomenology of Program Maintenance and Evolution, PhD thesis, Imperail College, 1981.

[18] C.F. Kemerer and S.Slaughter, An Empirical Approach to Studying Software Evolution, IEEE Trans. Soft. Eng. Vol 25,no. 4, July/August 1998, pp. 493-509.

[19] A. Antón and C. Potts, Functional Paleontology: System Evolution as the User Sees It, Proc. ICSE 2001.

[20] V. Nanda and N.H. Madhavji, the Impact of Environmental Evolution on Requirements Changes, Proc ICSE 2002

[21] A. Capiluppi, M. Morisio and J.F. Ramil, the Evolution of Source Folder Structure in Activiely Evolved Open Source systems, Metrics 2004 Symposium

[22] W. F. Opdyke, Refactoring Object-Oriented Framework, Thesis, University of Illinois at Urbana-Champaign, 1992

[23] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional,1999.

[24] T. Mens and T. Tourwe. A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2):126-162, February 2004

[25] R. Van Der Straeten, V. Jonckers, and T. Mens. Supporting model refactorings through behaviourinheritance consistencies. In Proc. Int'l Conf. UML 2004, volume 3273 of Lecture Notes in Computer Science, pages 305-319. Springer-Verlag, October 2004

[26] J. Zhang, Y. Lin, and J. Gray. Generic and domain-specific model refactoring using a model transformation engine. In Model-driven Software Development - Research and Practice in Software Engineering. Springer Verlag, 2005.

[27] T. D'Hondt, K. De Volder, K. Mens, and R.Wuyts. Co-evolution of object-oriented design and implementation. In Proc. Int'l Symp. Software Architectures and Component Technology. Kluwer Academic Publishers, January 2000.

[28] R.Wuyts. A LogicMeta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation. PhD thesis, Vrije Universiteit Brussel, January 2001.

[29] M. M. Lehman, D. E. Perry, and J. F. Ramil. On evidence supporting the feast hypothesis and the laws of software evolution. In Proc. Int'l Symp. Software Metrics. IEEE Computer Society Press, 1998.

[30] L. Williams and A. Cockburn. Agile software development: It's about feedback and change. IEEE Computer, 36(6):39-43, June 2003.

[31] M. Lehman and J. C. Fernandez-Ramil, Software Evolution, "Software Evolution and Feedback: Theory and Practice", Wiley, 2006

[32] V.T. Rajlich, Software evolution: a road map, In the proceeding of ICSM 2001.

[33] J. Greenfield and K. Short, Software factories: assembling applications with patterns, models, frameworks, and tools, Wiley, New York, 2004.

[34] S. Sendall and W. Kozaczynski, "Model Transformation: the heart and soul of model-driven software development", IEEE Software, Vol. 20, #5, September 2003.

[35] B. Selic, "The pragmatics of model-driven development" IEEE Software, Vol. 20, #5, September 2003

[36] OMG, MDA guide version 1.0.1, http://www.omg.org/cgi-bin/doc?omg/03-06-01, 2003

[37] S. Sendall and W. Kozaczynski, "Model Transformation: the heart and soul of model-driven software development", IEEE Software, Vol. 20, #5, September 2003.

[38] OMG. Unified Modeling Language (UML) version 1.5. http://www.omg.org/cgi-bin/doc?formal/03-03-01, 2003.

[39] OMG. XML Metadata Interchange (XMI) version 2.0, http://www.omg.org/cgi-bin/doc?formal/2003-05-02, 2003.

[40] J. Arlow and I. Neustadt,Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. Addison-Wesley Professional (December 22, 2003)

[41] Wegmann O. Preiss, “MDA in Enterprise Architecture? The Living System Theory to the Rescue...”, EDOC 2003.

[42] D. J. Velleman, How to Prove It: A Structured Approach 2ed., Cambridge University Press, 2006.

[43] C. F. Schaefer and C. Ussery, VHDL: Hardware Description and Design, Springer, 1989

[44] S. Clarke, W. Harrison, H. Ossher, P. Tarr, Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code, OOPSLA 1999.

[45] D. Batory, J.N.Sarvela, A. Rauschmayer, Scaling Step-Wise Refinement, IEEE Transactions on Software Engineering, Volume: 30 (6), 2004.

[46] J. Liu, D. Batory, and C. Lengauer, "Feature Oriented Refactoring of Legacy Applications”, ICSE 2006.

[47] E Yourdon, LL Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Inc

[48] J. Alves-Foss, D. Conte de Leon, and P. Oman, “Experiments in the Use of XML to Enhance Traceability Between Object-Oriented Design Specifications and Source Code”, Proceedings of the 35th Annual Hawaii International Conference on System Sciences.

[49] K.M. Anderson, S.A. Sherba, and W.V. Lepthien, “Towards Large-Scale Information Integration”, ICSE 2002.

[50] OMG. Object Constraint Language 2.0 (OCL), http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf

[51] Klaus Pohl, Software product Line engineering

[52] Ming-Jen Huang and Takuya Katayama, Using Responsibility Modeling and Rule-Based Approach for Product-Line Evolution, SPLC 2007, Doctoral symposium, Kyoto, 2007

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1995.

[54] Jess, Sandia National Laboratories, Jess. http://herzberg.ca.sandia.gov/jess/

[55] The Eclipse Foundation, Eclipse, Modeling Framework Project (EMF), http://www.eclipse.org/modeling/emf/

[56] F. Budinsky, D. Steinberg, E. Merks , R. Ellersick, and T. J. Grose, Eclipse Modeling Framework, Addison-Wesley Professional, 2003.

[57] The Eclipse Foundation, The Eclipse Graphical Editing Framework, http://www.eclipse.org/gef/

[58] The Eclipse Foundation, The Eclipse Graphical Modeling Framework (GMF), http://www.eclipse.org/modeling/gmf/

[59] D. S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley 2003

[60] J. Greenfield, K. Short, S. Cook, S. Kent, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2004.

[61] C. Burrows, G. George, and S. Dart, Configuration Management, Ovum Ltd., 1996.

[62] S. Dart, “Concepts in Configuration Management Systems,” Proceedings of the Third International Workshop on Software Configuration Management, ACM SIGSOFT, 1991, pp. pages 1–18.

[63] Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In Proc. OOPSLA/GPCE Workshop, 2004.

[64] Y.Kalfoglou., Exploring Ontologies, The Handbook of Software Engineering and Knowledge Engineering, vol.1: Fundamentals, pp.: 863-887, Scientific Publishing, first volume published on January 2002.

Publications

[1] Ming-Jen Huang and Takuya Katayama. Steering Model-Driven Development of Enterprise Information System through Responsibilities. Proc. WSMDEIS 2005 (Miami Beach, U.S.A, May 24-25, 2005). INSTICC Press, Portugal, 2005, 165-170.

[2] Ming-Jen Huang and Takuya Katayama. Steering Model-Driven Evolution by Responsibilities. Proc. IWPSE 2005.

[3] Ming-Jen Huang and Takuya Katayama, Using Responsibility Modeling and Rule-Based Approach for Product-Line Evolution, SPLC 2007, Doctoral symposium, Kyoto, 2007.

Appendix A: Use Cases

The development of the case study is started by using use cases to capture business processes of the three software systems. Business-MS, Medical-SS, and Shopping-WS have 22, 6, and 15 business processes respectively. In this appendix, we list the use cases of them.

A.1 Business-MS

Actor	ID	Business process
Customers	S-1	Create order
Customers	S-2	Modify order
Customers	S-3	Cancel order
Customers	S-4	Request catalog
Customers	S-5	Review orders
Sales person	S-6	Create quotation
Sales person	S-7	Modify quotation
Sales person	S-8	Create sales order from quotation
Sales person	S-9	Create invoice from sales order
Sales person	S-10	Create return document form sales order
Sales person	S-11	Manage customer accounts
Sales person	S-12	Manage product catalog
Procurement staff	S-13	Create purchase order
Procurement staff	S-14	Modify purchase order
Procurement staff	S-15	Cancel purchase order
Procurement staff	S-16	Create invoice/shippment receipt
Procurement staff	S-17	Manage suppliers
Inventory staff	S-18	Create inventory item

Actor	ID	Business process
Inventory staff	S-19	Adjust stocking quantity of inventory items
Inventory staff	S-20	Transfer inventory items
Inventory staff	S-21	Create shipping document from sales order
System users	S-22	Log in

A.2 Medical-SS

Actor	ID	Business process
Physician	S-1	Create BT order
Nurse	S-2	Modify BT Order
Nurse	S-3	Cancel BT order
Nurse	S-4	Create blood pickup document from BT order
Patient	S-5	Sign BT order consensus
Lab staff	S-6	Update testing results
System users	S-7	Log in

A.3 Shopping-WS

Actor	ID	Business process
Shop owner	S-1	Create shop
Shop owner	S-2	Modify shop
Shop owner	S-3	Create sales order
Shop owner	S-4	Modify sales order
Shop owner	S-5	Create invoice from sales order
Shop owner	S-6	Create return document form sales order

Actor	ID	Business process
Shop owner	S-7	Manage customer accounts
Shop owner	S-8	Create product item
Shop owner	S-9	Manage product item
Shop owner	S-10	Adjust stocking quantity of product items
Shop owner	S-11	Create shipping document from sales order
Customer	S-12	Create purchase order
Customer	S-13	Modify purchase order
Customer	S-14	Cancel purchase order
System users	S-15	Log in

Appendix B: Example output results of Jess

This appendix lists the output results of the example shown in 5.3. In the list, f-n is the identification given by Je Jess. The first list of f-0 to f-16 is the initial contents before realizing the business-process responsibilities in Figure 5-11. The second list of f-0 to f-18 is the contents that they are realized. One single PRU defined in Figure 5-12 is selected by the rules in Figure 5-13. This rule adds a new user-requirements responsibility to represent the responsibility that is instantiated for this realization. It can be noticed that f-8 in the first list is not shown in the second list. That is, f-8 is missing in the second list. This is because this responsibility has been realized therefore is removed by the rule shown in Figure 5-14. We also implemented a rule to notify which responsibility has no PRU for realization. It outputs a notification (i.e. `No PRU found to realize <Fact-18>`) as shown in this example which was formatted in bold and italic style.

```
Jess, the Rule Engine for the Java Platform
Copyright (C) 2006 Sandia Corporation
Jess Version 7.0p1 12/21/2006

f-0   (MAIN::initial-fact)
f-1   (MAIN::actor (id 0) (project BPMS) (name SalesPerson) (collection-
name SalesPeople) (propertyTypes Department) (propertyNames String))
f-2   (MAIN::actor (id 1) (project BPMS) (name Customer) (collection-name
SalesPeople) (propertyTypes Credit) (propertyNames Money))
f-3   (MAIN::document (id 0) (project BPMS) (name SalesOrder)
(collection-name SalesOrders) (propertyTypes ) (propertyNames ))
f-4   (MAIN::document (id 1) (project BPMS) (name ItemCatalog)
(collection-name SalesOrders) (propertyTypes ) (propertyNames ))
f-5   (MAIN::document (id 2) (project BPMS) (name Item) (collection-name
Items) (propertyTypes ) (propertyNames ))
f-6   (MAIN::document (id 3) (project BPMS) (name ShoppingCart)
(collection-name Items) (propertyTypes ) (propertyNames ))
f-7   (MAIN::process (id 0) (project BPMS) (name CreateSalesOrder))
f-8   (MAIN::responsibility (id 0) (project BPMS) (name nil) (from-pru
nil) (world BusinessProcesses) (process 0) (task List) (holder
SalesPerson) (receiver nil) (document ItemCatalog) (target-counter -1))
f-9   (MAIN::responsibility (id 1) (project BPMS) (name nil) (from-pru
```

```
nil) (world nil) (process 0) (task Select) (holder Customer) (receiver
nil) (document Item) (target-counter -1))
f-10   (MAIN::responsibility (id 2) (project BPMS) (name nil) (from-pru
nil) (world nil) (process 0) (task Hold) (holder SalesPerson) (receiver
nil) (document ShoppingCart) (target-counter -1))
f-11   (MAIN::responsibility (id 3) (project BPMS) (name nil) (from-pru
nil) (world nil) (process 0) (task Checkout) (holder Customer) (receiver
nil) (document ShoppingCart) (target-counter -1))
f-12   (MAIN::responsibility (id 4) (project BPMS) (name nil) (from-pru
nil) (world nil) (process 0) (task Create) (holder SalesPerson) (receiver
nil) (document SalesOrder) (target-counter -1))
f-13   (MAIN::pru (id 0) (world BusinessProcesses) (name pru-0) (source
0) (target-count 1))
f-14   (MAIN::collaboration (pru 0) (sequence 1) (target 1))
f-15   (MAIN::responsibility (id 0) (project nil) (name nil) (from-pru
nil) (world BusinessProcesses) (process nil) (task List) (holder nil)
(receiver nil) (document nil) (target-counter nil))
f-16   (MAIN::responsibility (id 1) (project nil) (name nil) (from-pru
nil) (world UserRequirements) (process nil) (task List) (holder nil)
(receiver nil) (document nil) (target-counter nil))
For a total of 17 facts in module MAIN.
FIRE 1 MAIN::set-target-counter-to-the-unrealized-responsibility f-8, f-
13, f-15, f-14, f-16
<== Activation: MAIN::set-target-counter-to-the-unrealized-
responsibility :  f-8, f-13, f-8, f-14, f-16
<== Activation: MAIN::no-pru--for-unrealized-responsibility :  f-8
 <=> f-8 (MAIN::responsibility (id 0) (project BPMS) (name nil) (from-pru
nil) (world BusinessProcesses) (process 0) (task List) (holder
SalesPerson) (receiver nil) (document ItemCatalog) (target-counter 1))
 ==> f-17 (MAIN::done-target (source 0) (done ))
==> Activation: MAIN::realize-unrealized-responsibility :  f-15, f-13, f-
15, f-14, f-16, f-17
==> Activation: MAIN::realize-unrealized-responsibility :  f-8, f-13, f-8,
f-14, f-16, f-17
==> Activation: MAIN::realize-unrealized-responsibility :  f-15, f-13, f-
15, f-14, f-16, f-17
==> Activation: MAIN::realize-unrealized-responsibility :  f-8, f-13, f-
15, f-14, f-16, f-17
FIRE 2 MAIN::realize-unrealized-responsibility f-8, f-13, f-8, f-14, f-16,
f-17
 ==> f-18 (MAIN::responsibility (id 101) (project nil) (name nil) (from-
pru 0) (world UserRequirements) (process nil) (task List) (holder nil)
(receiver nil) (document nil) (target-counter -1))
==> Activation: MAIN::no-pru--for-unrealized-responsibility :  f-18
<== Activation: MAIN::realize-unrealized-responsibility :  f-8, f-13, f-
15, f-14, f-16, f-17
 <=> f-8 (MAIN::responsibility (id 0) (project BPMS) (name nil) (from-pru
nil) (world BusinessProcesses) (process 0) (task List) (holder
SalesPerson) (receiver nil) (document ItemCatalog) (target-counter 0))
==> Activation: MAIN::realize-unrealized-responsibility :  f-8, f-13, f-
15, f-14, f-16, f-17
==> Activation: MAIN::realize-unrealized-responsibility :  f-8, f-13, f-8,
f-14, f-16, f-17
```

```
==> Activation: MAIN::retract-satisfied-responsibility :  f-8
<== Activation: MAIN::realize-unrealized-responsibility :  f-15, f-13, f-
15, f-14, f-16, f-17
<== Activation: MAIN::realize-unrealized-responsibility :  f-8, f-13, f-8,
f-14, f-16, f-17
<== Activation: MAIN::realize-unrealized-responsibility :  f-15, f-13, f-
15, f-14, f-16, f-17
<== Activation: MAIN::realize-unrealized-responsibility :  f-8, f-13, f-
15, f-14, f-16, f-17
 <=> f-17 (MAIN::done-target (source 0) (done 1))
FIRE 3 MAIN::retract-satisfied-responsibility f-8
 <== f-8 (MAIN::responsibility (id 0) (project BPMS) (name nil) (from-pru
nil) (world BusinessProcesses) (process 0) (task List) (holder
SalesPerson) (receiver nil) (document ItemCatalog) (target-counter 0))
FIRE 4 MAIN::no-pru--for-unrealized-responsibility f-18
No PRU found to realize <Fact-18>
FIRE 5 MAIN::no-pru--for-unrealized-responsibility f-12
No PRU found to realize <Fact-12>
FIRE 6 MAIN::no-pru--for-unrealized-responsibility f-11
No PRU found to realize <Fact-11>
FIRE 7 MAIN::no-pru--for-unrealized-responsibility f-10
No PRU found to realize <Fact-10>
FIRE 8 MAIN::no-pru--for-unrealized-responsibility f-9
No PRU found to realize <Fact-9>
 <== Focus MAIN
f-0   (MAIN::initial-fact)
f-1   (MAIN::actor (id 0) (project BPMS) (name SalesPerson) (collection-
name SalesPeople) (propertyTypes Department) (propertyNames String))
f-2   (MAIN::actor (id 1) (project BPMS) (name Customer) (collection-name
SalesPeople) (propertyTypes Credit) (propertyNames Money))
f-3   (MAIN::document (id 0) (project BPMS) (name SalesOrder)
(collection-name SalesOrders) (propertyTypes ) (propertyNames ))
f-4   (MAIN::document (id 1) (project BPMS) (name ItemCatalog)
(collection-name SalesOrders) (propertyTypes ) (propertyNames ))
f-5   (MAIN::document (id 2) (project BPMS) (name Item) (collection-name
Items) (propertyTypes ) (propertyNames ))
f-6   (MAIN::document (id 3) (project BPMS) (name ShoppingCart)
(collection-name Items) (propertyTypes ) (propertyNames ))
f-7   (MAIN::process (id 0) (project BPMS) (name CreateSalesOrder))
f-9   (MAIN::responsibility (id 1) (project BPMS) (name nil) (from-pru
nil) (world nil) (process 0) (task Select) (holder Customer) (receiver
nil) (document Item) (target-counter -1))
f-10   (MAIN::responsibility (id 2) (project BPMS) (name nil) (from-pru
nil) (world nil) (process 0) (task Hold) (holder SalesPerson) (receiver
nil) (document ShoppingCart) (target-counter -1))
f-11   (MAIN::responsibility (id 3) (project BPMS) (name nil) (from-pru
nil) (world nil) (process 0) (task Checkout) (holder Customer) (receiver
nil) (document ShoppingCart) (target-counter -1))
f-12   (MAIN::responsibility (id 4) (project BPMS) (name nil) (from-pru
nil) (world nil) (process 0) (task Create) (holder SalesPerson) (receiver
nil) (document SalesOrder) (target-counter -1))
f-13   (MAIN::pru (id 0) (world BusinessProcesses) (name pru-0) (source
0) (target-count 1))
```

f-14 (MAIN::collaboration (pru 0) (sequence 1) (target 1))
f-15 (MAIN::responsibility (id 0) (project nil) (name nil) (from-pru nil) (world BusinessProcesses) (process nil) (task List) (holder nil) (receiver nil) (document nil) (target-counter nil))
f-16 (MAIN::responsibility (id 1) (project nil) (name nil) (from-pru nil) (world UserRequirements) (process nil) (task List) (holder nil) (receiver nil) (document nil) (target-counter nil))
f-17 (MAIN::done-target (source 0) (done 1))
f-18 (MAIN::responsibility (id 101) (project nil) (name nil) (from-pru 0) (world UserRequirements) (process nil) (task List) (holder nil) (receiver nil) (document nil) (target-counter -1))
For a total of 18 facts in module MAIN.

Appendix C: Examples of PRU Data

C.1 PRUs for business-processes realization

World	Task	Target	Holder	Receiver
BusinessProcess	RequestCreation			
UserRequirements	NavigateToCreation		TargetSystem	
BusinessProcess	InformInput			
UserRequirements	ProvideInput		TargetSystem	
	ValidateInput		TargetSystem	
BusinessProcess	InformData			
UserRequirements	DisplayData		TargetSystem	
BusinessProcess	Process			
UserRequirements	Process		TargetSystem	
BusinessProcess	CreateData			
UserRequirements	CreateData		TargetSystem	
	DisplayDataCreationResult		TargetSystem	
BusinessProcess	Send			
UserRequirements	Send		TargetSystem	
BusinessProcess	GetData			
UserRequirements	GetData		TargetSystem	
	DisplayDataGetResult		TargetSystem	
BusinessProcess	RequestUpdate			
UserRequirements	NavigateToUpdate		TargetSystem	
BusinessProcess	CheckDataStatus			
UserRequirements	CheckDataStatus		TargetSystem	
	DisplayDataCheckResult			
BusinessProcess	ModifyData			
UserRequirements	ModifyData		TargetSystem	
	DisplayDataModifyResult		TargetSystem	
BusinessProcess	Ask			
UserRequirements	Query		TargetSystem	
	Browse		TargetSystem	
BusinessProcess	Acknowledge			

World	Task	Target	Holder	Receiver
UserRequirements	ProvideInput		TargetSystem	
	Validate		TargetSystem	
BusinessProcess	Cancel			
UserRequirements	Cancel		TargetSystem	
	Display		TargetSystem	
BusinessProcess	Provide			
UserRequirements	List		TargetSystem	
	ProvideBrowse		TargetSystem	
BusinessProcess	Agree			
UserRequirements	Agree		TargetSystem	
	Mark		TargetSystem	
BusinessProcess	Confirm			
UserRequirements	ProvideConfirm		TargetSystem	
	Mark		TargetSystem	
BusinessProcess	Input		InventoryStaff	
UserRequirements	Display		TargetSystem	
	ProvideInput		TargetSystem	
	ValidateInput		TargetSystem	

C.2 PRUs for user-requirements realization

World	Task	Target	Holder	Receiver
UserRequirements	NavigateToCreation		TargetSystem	
SoftwareDesign	NavigateTo		PageController	
UserRequirements	ProvideInput		TargetSystem	
SoftwareDesign	ProvideInput		PageController	
UserRequirements	ValidateInput		TargetSystem	
SoftwareDesign	ValidateInput		PageController	
UserRequirements	DisplayData		TargetSystem	
SoftwareDesign	DisplayData		PageController	
UserRequirements	Process		TargetSystem	
SoftwareDesign	Service		BusinessDelegate	
	Process		BusinessService	
UserRequirements	CreateData		TargetSystem	

World	Task	Target	Holder	Receiver
SoftwareDesign	ServiceDataCreation		BusinessDelegate	
	CreateData		BusinessService	
	StoreData		DataAccess	
UserRequirements	DisplayDataCreationResult		TargetSystem	
SoftwareDesign	DisplayDataCreationResult		PageController	
UserRequirements	Send			
SoftwareDesign	Service		BusinessDelegate	
	Send		BusinessService	
	Retrieve		DataAccess	
	MailServiceResult		BusinessService	
UserRequirements	GetData		TargetSystem	
SoftwareDesign	ServiceDataGet		BusinessDelegate	
	GetData		BusinessService	
	RetrieveData		DataAccess	
UserRequirements	DisplayDataGetResult		TargetSystem	
SoftwareDesign	DisplayDataGetResult		PageController	
UserRequirements	ModifyData		TargetSystem	
SoftwareDesign	ServiceDataModification		BusinessDelegate	
	ModifyData		BusinessService	
	UpdateData		DataAccess	
UserRequirements	DisplayDataModificationResult		TargetSystem	
SoftwareDesign	DisplayDataModificationResult		PageController	
UserRequirements	NavigateToUpdate		TargetSystem	
SoftwareDesign	NavigateTo		PageController	
UserRequirements	CheckDataStatus		TargetSystem	
SoftwareDesign	ServiceCheckDataStatus		BusinessDelegate	
	CheckDataStatus		BusinessService	
UserRequirements	DisplayDataCheckResult		TargetSystem	
SoftwareDesign	DisplayDataCheckResult		PageController	
UserRequirements	Cancel		TargetSystem	
SoftwareDesign	Service		BusinessDelegate	
	Cancel		BusinessService	
UserRequirements	List		TargetSystem	
SoftwareDesign	List		PageController	
	Service		BusinessDelegate	

World	Task	Target	Holder	Receiver
	GetData		BusinessService	
UserRequirements	Mark		TargetSystem	
SoftwareDesign	Input		PresentationLayer	
	Mark		BusinessLayer	
	Display		PresentationLayer	

C.3 PRUs for software-design realization

World	Task	Target	Holder	Receiver
SoftwareDesign	ServiceDataGet		BusinessDelegate	
ProgramConstruction	GetData			
	LookupService			
SoftwareDesign	GetData		BusinessService	
ProgramConstruction	GetData			
SoftwareDesign	RetrieveData		DataAccess	
ProgramConstruction	CreateDAO			
	RetrieveData			
	AccessRepository			
SoftwareDesign	DisplayDataGetResult		PageController	
ProgramConstruction	DisplayDataGetResult		JSPTag	
SoftwareDesign	NavigateTo		PageController	
ProgramConstruction	Link		JSPTag	
SoftwareDesign	ProvideInput		PageController	
ProgramConstruction	ProvideInput		JSPTag	
SoftwareDesign	ValidateInput		PageController	
ProgramConstruction	ValidateInput		JSPTag	
SoftwareDesign	DisplayData		PageController	
ProgramConstruction	DisplayData		JSPTag	
SoftwareDesign	Service		BusinessDelegate	
ProgramConstruction	ExecuteService		BusinessDelegate	
	LookupService			
SoftwareDesign	Process		BusinessService	
ProgramConstruction	Process			
SoftwareDesign	ServiceDataCreation		BusinessDelegate	

World	Task	Target	Holder	Receiver
ProgramConstruction	CreateData			
	LookupService			
SoftwareDesign	CreateData		BusinessService	
ProgramConstruction	CreateData			
SoftwareDesign	StoreData		DataAccess	
ProgramConstruction	CreateDAO			
	StoreData			
	AccessRepository			
SoftwareDesign	DisplayDataCreationResult		PageController	
ProgramConstruction	DisplayDataCreationResult		JSPTag	
SoftwareDesign	Service		BusinessDelegate	
ProgramConstruction	SendConfirm			
	LookupService			
SoftwareDesign	Send		BusinessService	
ProgramConstruction	GetDocument			
SoftwareDesign	SelectData		DataAccess	
ProgramConstruction	CreateDAO			
	SelectData			
	AccessRepository			
SoftwareDesign	Send		BusinessService	
ProgramConstruction	GenerateConfirm			
	SendConfirm			
SoftwareDesign	ServiceDataList		BusinessDelegate	
ProgramConstruction	ListData			
	LookupService			
SoftwareDesign	ExecuteListData		BusinessService	
ProgramConstruction	ExecuteListData			
SoftwareDesign	ListData		DataAccess	
ProgramConstruction	CreateDAO			
	RetrieveData			
	AccessRepository			
SoftwareDesign	DisplayDataListResult		PageController	
ProgramConstruction	DisplayDataListResult		JSPTag	
SoftwareDesign	ServiceCheckDataStatus		BusinessDelegate	
ProgramConstruction	CheckDataStatus		BusinessDelegate	

World	Task	Target	Holder	Receiver
	LookupService		ServiceLocator	
SoftwareDesign	ExecuteCheckDataStatus		BusinessService	
ProgramConstruction	ExecuteCheckDataStatus		BusinessService	
SoftwareDesign	DisplayDataCheckResult		PageController	
ProgramConstruction	DisplayDataCheckResult		JSPTag	
SoftwareDesign	ExecuteCancelService		BusinessDelegate	
ProgramConstruction	ExecuteCancelService		BusinessDelegate	
	LOokupService		ServiceLocator	
SoftwareDesign	ExecuteCancel		BusinessService	
ProgramConstruction	CancelDocument		BusinessService	
SoftwareDesign	List		PageController	
ProgramConstruction	ListDocuments		JSPTag	
SoftwareDesign	ExecuteGetCatalogService		BusinessDelegate	
ProgramConstruction	GetCatalog		BusinessDelegate	
	LookupService		ServiceLocator	
SoftwareDesign	Get		BusinessService	
ProgramConstruction	GetDocument		BusinessService	
SoftwareDesign	ExecuteMarkService		BusinessDelegate	
ProgramConstruction	AgreePayment		BusinessDelegate	
	LookupService		ServiceLocator	
SoftwareDesign	Mark		BusinessService	
ProgramConstruction	MarkProperty		BusinessService	
SoftwareDesign				
ProgramConstruction				