Architecture-Based Problem Frames Constructing for Software Reuse

CHU Wang
Dept. of Computer Science, Xi’an
Jiaotong University
Xi’an, P. R. China.
+86-536-8268409
chuw1119@163.com

QIAN Depei
Dept. of Computer Science, Xi’an
Jiaotong University
Xi’an, P. R. China.

LIU Chuda
Dept. of Computer Science, Xi’an
Jiaotong University
Xi’an, P. R. China.

ABSTRACT
Reusability is not universal property of program codes. The problem frames can support the software component reuse effectively. The problem frames approach provides an opportunity for software developers to gain experience and expertise for problem domain. This paper proposes architecture-based problem frames constructing process that results in traceable problem patterns (business patterns and requirements patterns), and the traceability between problem space and software requirements is established as the side effect of the problem frames constructing process. The architecture-based problem frames constructing can integrate with architecture-based software design process seamlessly. The relationships between context and requirements are identified explicitly, so that the problem frames are traceable and reusable.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications—Methodologies; D.2.13 [Software Engineering]: Reusable Software—Reuse models.

General Terms
Design, Documentation.

Keywords
Problem frames approach, Software reuse, Architecture-based modeling, Patterns.

1. INTRODUCTION
What makes software development so difficult? Kyle Eischen states in [1] that traditional software engineering techniques focus from the inception on the system that needs to be built, little effort is given to analyze and understand how the business in which the system is to be deployed is organized and how it operates. Many projects fail because the developer does not truly understand the real requirements of the users and their environment. Brian Lawrence et al. put forward the top risks of requirements engineering in [2], the most of risks are related to the problem domain understanding. Software engineering should begin with context and problem analysis to determine the boundary of the software system as well as the goals of the system.

Currently, the problem frame has become one of the vital techniques in software engineering for requirements analysis and problem domain specifications. A problem frame defines the shape of a problem by capturing the characteristics and interconnections of the parts of the world it is concerned with. Problem frames provides a means of analyzing and decomposing problems [3].

The problem frames approach can play important role in reuse-centric software development methods (e.g., software product lines approach [4]), since problem-oriented software reuse is more effective than solution-oriented reuse. Two important lessons from component-based software reuse have been learned: 1) opportunistic reuse of components does not work and component reuse requires a managed, systematic, and explicit effort. All the development stages from problem analysis to system assembly should support reuse strategy, and 2) reusability is not universal property of program codes and it rather depends on the context of the problem that is relatively stable. Bottom-up reuse has proven not to work in practice, for component integration, problem frames and design must be shared and reused before component reuse.

An increasing number of organizations are realizing that most software development projects are not one-of-a-kind efforts, they can develop core assets for future contracts and products in the same application domain. Hence it is necessary to explicitly distinguish between software development for reuse that creates reusable core assets and software development with reuse that uses core assets to create products.

The core assets must be easy to trace, to communicate to stakeholders, and to maintain. Because different development teams are responsible for core assets development and product development, traceability between problem domain and solution domain is essential to understand and to reuse the core assets. Software maintenance profits from traceability because the maintainers understand why a system was built the way it was, and can better assess the impact of requirement or business modifications. Product assembly profits from traceability,
engineers can retrieve and select the components from the core assets that meet stakeholders’ needs, and then assemble them.

In order to support the problem-oriented software reuse, in this paper, we combine architecture-based modeling technique with problem frames constructing. The traceability between context and requirements can be captured and maintained as a side effect of the constructing process of problem frames.

Architecture defines a set of components, specifies a topological pattern for their interconnection, and imposes constraints on them. Architecture pattern is a way of reuse abstract knowledge about a recurrent problem in a particular context and its solution. When the problem is complex, it can be divided into a few more simple problems, each one is resolved in an independent pattern. The simplicity of a pattern and its “small size” make it easy to understand, to integrate, and to reuse. Pattern is “know-how” oriented that provides problem, solution, and engineering guidance to reuse them. Other forms of reusable components (i.e., software components) are “know” oriented [5]. The context and problem of the pattern make it easy for the developers to retrieve the right solution. Domain specific pattern is a natural way for the formulation of accumulated development expertise in analysis, design, and implementation phases. In this paper, an architecture pattern is represented as components and relationships among them.

This paper is organized as follows. In section 2, we outline the related works on domain analysis method of software product-line and architecture-based software development methods. Section 3 describes the architecture-based problem frames constructing process for software reuse. Section 4 gives a case study. Summary is given in section 5.

2. RELATED WORK
Feature-Oriented Domain Analysis (FODA) is a domain analysis methodology developed by the Software Engineering Institute (SEI). The FODA method is based on identifying features common to a class of systems. Features are user-visible characteristics of a system and organized into a tree of AND/OR nodes to identify the commonalities and variability within the system.

Gary Chastek et al. combine OOA with FODA in [6] to elicit, analyze, specify, and verify the requirements for a product-line. It consists of three phases: context analysis, domain modeling, and architecture modeling. The work products of context analysis are structure diagram and context diagram. The results of domain modeling include feature model, classes/objects diagram, state transition diagram, and standard vocabulary. During architecture modeling, process interaction model and module structure chart are produced.

Existing domain analysis approaches face a number of issues [7][8]:

- Lack of traceability between problem and software components, it is unclear which specific problem is solved by implemented component(s).
- Lack of explicit identification and modeling of the relationships among components results in lack of understanding of the core assets.
- Lack of structured description, the core assets may be misunderstood because of their large size and complexity.

Architecture has recently recognized as a new and effective way to improve our ability to reuse software components. MEI Hong et al. proposed ABC (Architecture Based Component oriented) approach that introduced the architecture into each phase of software lifecycle in order to keep traceability among software development phases [9].

Felix Bachmann et al. put forward the architecture-based design method (ABD) in [10]. The ABD method proceeds by recursively decomposing the system to be designed. The first decomposition is of the system; subsequent decompositions are refinements of the prior decomposition. At each decomposition step, the functional requirements are met by assigning responsibilities to the divisions of the element being decomposed. The quality and business requirements for that element are met by choosing an architecture style that describes how the divisions relate to one another.

In order to enhance the applicability of problem frames in architecture-based software development, Jon G. Hall et al. allow architectural structures and services to be considered as parts of the problem domain, relate software requirements with architectures by problem frames [3].

3. ARCHITECTURE BASED PROBLEM FRAMES CONSTRUCTING
The problem frames approach applies two interlocking principles. The first one is that a problem should be decomposed into sub-problems that can be developed by the applicable techniques. The second one is that sub-problem composition is demanding explicit consideration. The composition of the sub-problems must be shown to approximate closely enough to the original problem [11].

In this paper, context analysis and requirements analysis form the backbone of the architecture-based problem frames constructing process. By architecture-based problem frames, product developers can specify “what” a problem is rather than “how” to realize the desired solution in terms of the techniques of a particular implementation platform. Product assembly can be conducted by tracing and selecting software components based on problem frames.

The architecture-based problem frames constructing process follows the traditional “divide-and-conquer” approach of defining architecture. The problem is broken into consecutively more and more concrete and small sub-problems that can be individually tackled. Every phase of this process is a divide-and-conquer procedure that consists of three activities: decomposing, architecture design, and validating.

- Decomposing: The objective of this activity is to divide the goals or requirements into a number of sub-goals and assign them to components (sub-problems). In this activity, “Goals-Assignment” relationships between problem requirements and components are created, and they are called γ-relationships.
- Architecture design: The objective of this activity is to compose components into architecture. Determining coordination rule(s) of components is major work of this activity. “Part-Whole”
relationships between component and architecture are created, and they are called $\beta$-relationships.

- Validating: The objective of this activity is to check whether the decomposition is acceptable. In this activity, “Satisfied-By” relationship between problem requirements and architecture is created, and it is called $\lambda$-relationship.

The divide-and-conquer procedure results in a pattern. The created relationships are used to trace and to understand the architectures. A pattern is written as $P \rightarrow \{A | C_{i} \leq n\}$, where $P$, $C$ and $A$ represent “Problem requirements”, “Component” and “Architecture” respectively. The problem requirements may be contained in an abstract component coming from high-level architecture. The architecture-based problem frames constructing creates two kinds of patterns: context pattern and requirements pattern.

### 3.1 Context Analysis

The notion of underlying the architecture-based problem frames constructing for software reuse is that a domain or business area responds to the same types of business events and similar business rules to ensure that the organization’s goals are achieved. Each business area has its own vocabulary, procedures, and approaches for allocating resources to fulfill its missions. The domain specific knowledge can be reused to understand and to describe a particular business environment when developers assemble products. It is necessary for the business model to be established prior to the definition of software requirements.

The objective of context analysis is to construct domain specific business pattern. It consists of three activities: business operation decomposing, business pattern constructing, and business pattern validating.

By analyzing the business operation and structure of organizations in a business area, domain engineers construct generic business patterns. The business analysis results in hierarchical patterns. When the business patterns are finished, domain engineers should check whether the business patterns truly represent the business structure, goals, rules, and constrains.

The role of the business patterns in problem frames based software reuse has three aspects. Firstly, it can be used as a common foundation for product developers to understand a business area. Secondly, it can be used as context of software requirements. User’s requirements have meaning only when referred to their operation context in which the users perform their tasks and in which they have problem and expect a solution [12]. Building business patterns and situating the user’s requirements in that context has become one of the key techniques in requirements engineering for analyzing and constructing a common understanding of user’s problem. Context is very important for requirements reuse, when decide whether or not to reuse pre-defined requirements. Developers have to understand the context that prompted the original designer to formulate these requirements. Thirdly, they can be reused to describe user’s business in application engineering, which can reduce the time spent on business understanding. Experience shows that business pattern has a very high stability and life expectancy.

### 3.2 Requirements Analysis

Although requirements refer to specific domain and to specific problem, but requirements within the same domain or similar problem are more likely to be similar than the software components. In software development process, the hardest task is to determine software requirements. Requirements reuse can provide significant cost savings and quality improvement. Requirements reuse can provide an enhanced opportunity for reusing the software components. Moreover, when the reusable components are identified “at the source”, the reuse is optimized [5].

Existing requirements analysis approaches don’t support reuse strategy adequately due to the lack of traceable and understandable requirements model. Providing the traceability between business goals and software requirements is identified as one of the most important requirement engineering problems [13]. In this paper, we construct requirements model by architecture-based modeling method. The links between context and requirements are described explicitly. Business patterns are used as context of software requirements.

Architecture-based requirements modeling aims on two aspects: a) the constructed requirements model is understandable, and b) it can integrate with the architecture-based design approach seamlessly, which results in traceable core assets (from software requirements to software components), so that the software components can be retrieved and assembled easily.

The requirements analysis consists of three activities: requirements partitioning, requirements modeling and requirements validating.

- Requirements partitioning. Developers have to determine a partition of requirements between user and software, that is, the boundary of software system is determined.

- Requirements modeling. The objective is to define the coordination rule(s) of the user and the software system, that is, to describes how the user to interact with the software system. The result of this activity is architecture-based requirement patterns that contain software requirements and business goals.

- Requirements validating. When the requirement patterns are finished, it is time for the developers to validate whether the requirements meets the business goals.

In software development with problem frames based software reuse, when developers reuse the business patterns to describe user’s business environment, the requirements can be retrieved by the business goals and the requirements pattern, and can be assembled into the product requirements.

The architecture-based problem frames constructing for software reuse results in a traceable hierarchical pattern model (problem frames).

When developers understand and reuse problem frames, the following questions have to be answered:

- Why do the requirements exist?
- What links do exist between the business goals and the requirements?
- How do the requirements meet the business goals?
Every requirement of the problem frames should have specific reason for existence, which can be traced, either to some requirements or to some context. Traceability focuses on how to trace between the context and the requirements to understand the implications of a specific requirement. γ-relationship shows why a requirement exists. β-relationship and λ-relationship help developers to trace how the requirements meet the business goals.

Due to the limited space, this paper only outlines software development with problem frames based software reuse briefly. Software reuse occurs firstly at the application analysis phase in which the business patterns are reused to model the application environment, and then requirements patterns, design patterns, and software components are traced and assembled. At every phase, the constructed architecture is validated. Architecture-based problem frames can support software reuse by identifying and modeling the links between the business goals and the requirements explicitly.

Newsroom$_0$ is a high level view that only contains business goal that is decomposed into three sub-goals, and they are assigned to Editor, Director, Proof_reader respectively.

Editor, Director, and Proof_reader cooperate with each other to achieve business goal of newsroom. Newsroom is an architecture that contains the coordination rule (business procedure).

Fig.1 also contains a requirements pattern, written as Editor$_e$→$\gamma _{11}$ Editor$_c$ {Editor$_e$, Editors$_s$}, Editor$_e$ contains the responsibilities of the user. Editors$_s$ contains the software requirements. Editor$_c$ contains the coordination rules that describe the interactions between the user and the software system.

When architecture Newsroom is reused to describe a particular newsroom, Editor$_s$ can be traced by component Editor and requirements pattern Editor$_e$→$\gamma _{11}$ Editor$_c$ {Editor$_e$, Editors$_s$}.

In Fig.1, if component Editor is given, then $\gamma _{11}$: Newsroom$_e$→Editor explains why Editor exists, and pattern Editor$_e$→$\gamma _{11}$ Editor$_c$ {Editor$_e$, Editors$_s$} describes how the Editor’s business goal is achieved. If Editor is changed, $\gamma _{11}$, $\gamma _{12}$, $\beta _{11}$, and $\lambda _{11}$ can be used to estimate the ripple effects of the change.

5. SUMMARY AND FUTURE WORK

Architecture has gained importance in software development process as a powerful means of system abstraction. Architecture pattern expresses a fundamental structural organization for the software system or business system by providing a set of components and the relationships among them. In this paper, the architecture-based problem frames constructing is proposed. Architecture patterns answer such questions as “why do the requirements exist?”, “what relationships are there between the business goals and the software requirements?”, and “how the business goals are achieved?”. It is well understandable, hence acceptable by the customer and the developer. When a specific business is mapped to corresponding business pattern, the requirements patterns can be retrieved and assembled easily due to the traceability between the business patterns and the requirements patterns. Reuse of problem frames in software development can be seen as a pattern selection and pattern composition process. Architecture-based problem frames constructing process can be seamlessly integrated with the architecture-based design process by which the traceability between the business goals and the software components can be established easily, so that the software components can be retrieved and assembled efficiently.

Future work will focus on formal definition of the relationships between the business goals and the requirements in order to support product assembly and test by means of automated tool.

6. REFERENCES


