Mapping requirements to software architecture by feature-orientation

Dongyun Liu, Hong Mei
Institute of Software, School of Electronics Engineering and Computer Science
Peking University, Beijing 100871, P.R.China
liudy@cs.pku.edu.cn, meih@pku.edu.cn

Abstract

Requirements engineering and software architecting are two key activities in software life cycle. Researchers have paid much attention to mapping and transformation from requirements to software architecture, but there's still lack of effective solutions. In this paper, the inadequacy of traditional mapping approaches (such as approaches in structured method and OO method) for this challenge is analyzed, and further a feature-oriented mapping approach is introduced. The rationale, process and guidelines for this approach are specified, and the approach is illustrated by an example of bank account and transaction (BAT) system.

1. Introduction

Requirements engineering and software architecting are two important activities in software life cycle. Requirements engineering is concerned with purposes and responsibilities of a system. It aims for a correct, consistent and unambiguous requirements specification, which will become the baseline for subsequent development, validation and system evolution. In contrast, software architecting is concerned with the shape of the solution space. It aims at making the architecture of a system explicit and provides a blueprint for the succeeding development activities. It is obvious that there exist quite different perspectives in user (or customer) requirements and software architecture (SA). Mapping from requirements to SA is by no means trivial work. In traditional software development methods, the mapping relationship between requirements and SA is indirect and not straightforward, and existing mapping solutions are inadequate for mapping user (or customer) requirements to SA. In order to adapt to iterative, incremental and evolutionary development paradigm, it is necessary to make the mapping relationship between user (or customer) requirements and SA direct and straightforward, so as to support the traceability between requirements and SA more effectively.

As we have noticed, today more and more researchers pay their attentions to the research of feature-oriented software development methods. There have been efforts to apply feature to software development. In 1982, Davis [1] identified features as an important organization mechanism for requirements specification. In 1990 Kyo C. Kang [2] etc. proposed feature-oriented domain analysis (FODA) method. In this method, the concept of using feature model for requirements engineering was introduced. As a main activity in domain modeling, feature analysis is intended to capture the end-user's (and customer's) understanding of the general capabilities of applications in a domain. Later, FORM method [3] extends FODA to the software design and implementation phases and prescribes how the feature model is used to develop domain architectures and components for reuse. FORM method is quite fit for

software development in mature domain where standard terminology, domain experts and up-to-date documents are available. C. Reid Turner [4] puts forward a conceptual framework for feature engineering in 1999. Turner prefers to look feature as an important organizing concept within the problem domain and proposes carrying a feature orientation from the problem domain into the solution domain. Turner's framework comes from software development experience in telecommunication domain, and is still conceptual and incomplete. It does not provide particular solution for mapping requirements to SA from software engineering perspective. But above researches and practice show that it is feasible and effective to make features explicit in software development and to take feature orientation as a paradigm during the software life cycle.

In this paper, we will explore how to apply feature orientation as a solution for the mapping problem between requirements and SA from general software engineering perspectives, focusing on the mapping and transformation process. Our solution is to organize requirements in problem domain into a feature model, and then base our architectural modeling on the feature model, with the goal maintaining direct and natural mapping between requirements model and architecture models. Further, we will address functional features and nonfunctional features separately in different architectural models. Our approach is not a replacement of but an improvement on traditional methods. Our approach can integrate closely with OO method. The modeling concepts and notation adopted in this paper are based on UML, but have appropriate extension.

The rest of this paper is organized as follows. Section 2 analyzes the relationship between requirements engineering and software architecting, and specifies the necessity for supporting traceability between requirements and SA. Section 3 analyzes the inadequacy of mapping approaches in traditional methods. Section 4 proposes a feature-oriented mapping solution, and specifies the rationale, process and guidelines for this approach. Section 5 concludes the paper and further research effort is envisioned. Application of our mapping approach to the bank accounts and transactions system (BAT) has been used in this paper as an illustrative example.

2. Requirements Engineering and Software Architecting

The IEEE standard [5] defines "requirement" as

"(1) A condition or capability needed by a user to solve a problem or achieve an objective.

(2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document.

(3) A documented representation of a condition or capability as in (1) or (2)."

This definition is not so clear. In practice, requirements for a software system may exist in multiple different abstract levels, varying from organization's business requirements, through user's task requirements, to eventual software requirements specification (SRS).

Requirements engineering aims at reaching a good understanding of the problem domain and user's (or customer's) needs through effective problem analysis techniques, and producing a correct, unambiguous, complete and consistent requirements specification which serves as a baseline for developers to implement the software system. Requirements engineering only focuses on problem domain and system responsibilities, but not design and implementation details.

SA has become an important research field for software engineering community. There exists a consensus that for any large software system, its gross structure-that is, its high-level organization of computational elements and interactions between those elements-is a critical aspect of design [6][7]. It is widely accepted that SA is a very important product and software architecting is a necessary phase in software life cycle. As an important design concept, SA "can serve as the key milestone in the entire software life cycle process". SA's "support of the needs of system engineers, customers, developers, users, and

maintainers, also implies that is involved in all phases of the software and system life cycle”[8].

Until now software engineering researchers don’t reach an agreement about the relationship between requirements engineering and software architecting. Following waterfall development model, software architects should not begin software architecting until a complete, correct and consistent requirements specification is reached. But some researchers[10] have pointed out that this model is discredited. In “multilevel life cycle chart” model, proposed by Merlin Dorfman [10], requirements engineering is involved throughout the software architecting process, that is, the steps of requirements analysis and design alternate. Rational Software Corporation [9] proposes the Unified Process, which is a use case driven, architecture-centric, and iterative and incremental process framework. In spite of existing different perspectives, now iterative, incremental, evolutionary and concurrent development paradigms are gaining more and more wide-spread acceptance. In development following such paradigms, it is more important to maintain direct and natural mapping and traceability between requirements specification and SA.

3. Traditional mapping approaches

Looking back on the development of software development methodology, it is not difficult to find that keeping the traceability and the consistency in concepts between requirements and designs always are the goals that we pursue. Two main software development methods, structured method and object-oriented method, both provide solutions for mapping analysis model to design model.

In structured method, software requirements are captured in Data Flow Diagram (DFD), and design is described in Module Structure Chart (MSC). Because there exists evident difference between the basic concepts and principles of DFD and MSC, mapping DFD to MSC is difficult and just by heuristic rules. Object-oriented approach cured the symptom that the structured paradigm did not. Because Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) use the uniform basic concepts and principle, the mapping between OOA model and OOD model is natural and direct. Also, keeping traceability is easy and transformation could be done mechanically.

But both structured method and OO method don’t provide complete solution for mapping requirements to SA indeed. On one hand, in both methods, SA and components are not paid enough attention to; On the other hand, both DFD and OOA model describe internal structure of the system from developer’s view and not external behavior from end users’ view. They contain some information that is not of interest to end-users (or customers). So there is still a gap between analysis model (DFD or OOA model) and user requirements description. Based on above analysis, we can conclude that, the mapping approaches in traditional methods are inadequate for mapping from requirements to SA.

4. Feature-oriented mapping

In this section we will explore how to apply feature orientation as a solution for mapping and transformation from requirements to SA aiming at improving traditional methods.

Feature has been defined in various ways, some important definitions are as follows: A feature is “a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective”[4]; A feature is “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems”[2]; A feature is “an externally desired service by the system that may require a sequence of inputs to effect the desired result” [11]; A feature is “a service that the system provides to fulfill one or more stakeholder needs”[12]. We think that a feature is a higher-level abstraction of a set of relevant detailed software requirements, and is perceivable by users (or customers). So it is reasonable to identify features as “first-class entities” in requirements modeling, and

combine feature modeling and traditional requirements modeling together to describe requirements in different levels. Further, in architectural modeling we will take feature orientation as a goal, and try to maintain direct and natural mapping between feature model and architectural models at higher levels. By feature-orientation, we aim at making the mapping relationship between requirements and SA straightforward, which is impossible by traditional approaches.

We also have recognized the different effect of functional features and nonfunctional features on software architecture and address them separately. First, we get an initial partition of the proposed system into components based on functional features. Then, further optimization and transformation can be imposed on the partition, iteratively and incrementally, considering nonfunctional features.

The feature-oriented mapping process consists of two stages: feature-oriented requirements modeling and feature-oriented architectural modeling.

4.1 Feature-oriented requirements modeling

Feature-oriented requirements modeling is intended to capture users' (or customers') high-level expressions of desired system behavior in terms of application features, analyze the relationship between features, and then organize and refine the features into a feature-oriented requirements specification. Feature-oriented requirements modeling can be divided into three activities: feature elicitation, feature organization and analysis and feature refinement.

Feature elicitation

Features elicitation focuses on eliciting user requirements in terms of features. Keeping elicitation at abstract feature levels, we can avoid plunging into detailed functional requirements too early. Also, as the user often has expertise in the domain and knows the value of the features, problem analysis effort can be concentrated on user-desired features and unnecessary work can be reduced. Users' (or customers') knowledge about problem domain is main source of features. Books, user manuals, etc. are also sources of features. Main feature elicitation techniques include interview, questionnaire, requirements workshop, and so on. In mature domains, analyzing the terminology of the domain language is also an effective and efficient way to identify features.

Feature analysis and organization

As potential features are identified and elicited, they are analyzed and organized into a feature hierarchy in a tree form. The features collected can be divided into functional features and nonfunctional features. All features reflect stakeholders' need to solve their problems. According Karl E. Weigers' view [13], stakeholders' requirements may exist in multiple levels, including business requirements, user requirements and functional requirements. As abstraction of functionality, features may exist at either business level or user level. A feature at business level describes the high-level desire of an organization or a customer for future system. Features at user level describe services which future system should provide for user tasks and constraints on the services. We first partition the features into the two levels, and we then further organize the features based on following criteria:

- The features to support a specific business process can be grouped and abstracted as a higher-level feature
- The features to support a specific user class can be grouped and abstracted as a higher-level feature
- A nonfunctional feature that is a constraint on a functional feature becomes a sub-feature of the functional feature.
- If a feature at user level is used to realize a feature at business level, then the former becomes a sub-feature of the latter. For instance, in the bank account and transaction system (BAT) example (see Figure 1), "identify client" feature is a realization of the nonfunctional feature "security", so "identify client" feature becomes a sub-feature of "security" feature.

There exist various relationships among the features. We

have identified several kinds of relationships: “composed-of”, “generalization/specialization”, “derived-from”, “constrained-by” and “dependent-on”, etc. As shown in Figure 1, “identify client” is derived from “security”, “withdraw money” is constrained by “response time <= 1min”, all customer services is dependent on “identify client”.

Features themselves may be “mandatory” or “optional”. A mandatory feature is necessary for general users, and an optional feature is necessary for partial users. For example, “withdraw money” is a “mandatory” feature, but “transfer money” is an “optional” feature.

Feature refinement

Now we have a feature hierarchy, but it is not specific enough for implementation. The next task is to refine the features into detailed functional requirements. Here use case technique can be used to elaborate a feature into a set of functionality.

Figure1 presents the resulting requirements model through feature-oriented modeling.

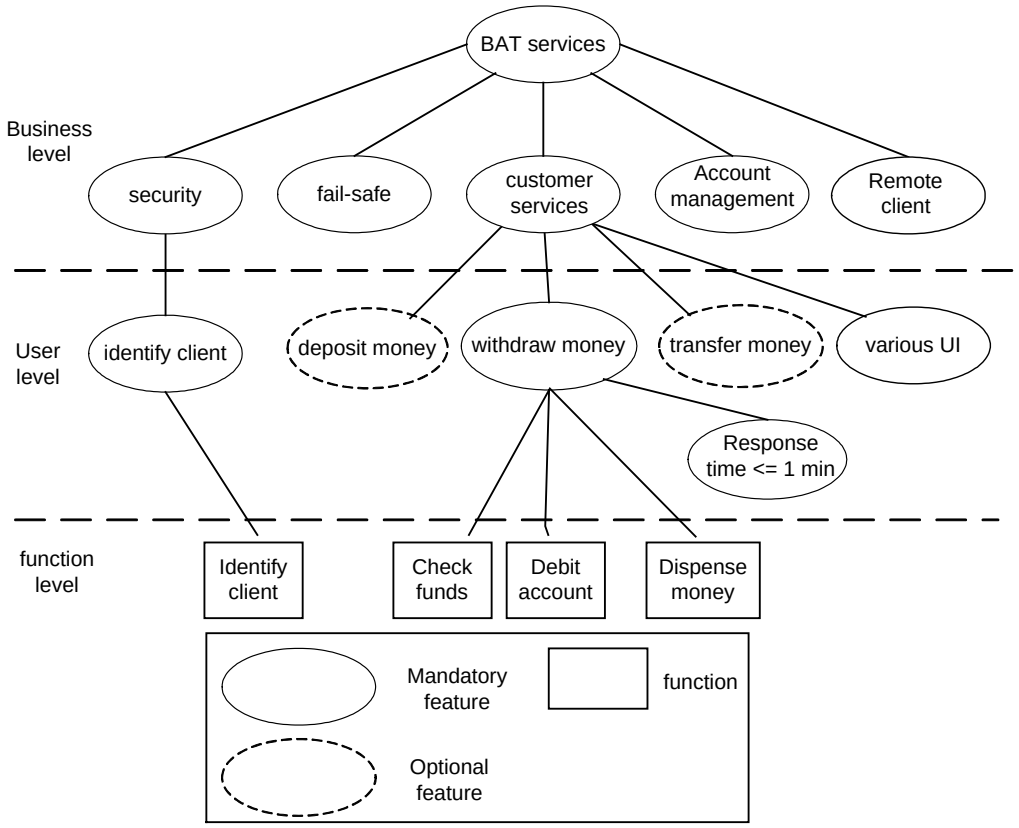

Figure 1. The feature model of BAT system

4.2 Feature-oriented architecture modeling

After we have got requirements specification organized by features, we can take it as an input to architecture modeling and derive SA from it. We will take feature prominence as a goal and try to maintain direct and natural mapping between feature model and architecture models. Also, we have recognized that functional features and nonfunctional features have each different contribution to SA. A functional feature can be mapped to a subsystem or a component. Nonfunctional features can generally not be pinpointed to a particular component, but have impacts on the system structure and behavior as a whole. So we can address them separately in different models. We define SA

from three different viewpoints: conceptual architecture, logical architecture, and deployment architecture. As shown in figure 2, conceptual architecture focuses on the system's partition based on functional features, not considering nonfunctional features. Logical architecture focuses on logic design for addressing nonfunctional features, considering the implementation environment. Deployment architecture focuses on physical distribution of the system, addressing related nonfunctional features.

Conceptual architecture

The conceptual architecture identifies the system components, the responsibilities of each component, and relationships between components in terms of application domain concepts. It tries to preserve the structure of problem domain by partitioning system based on functional features and problem domain structure. Each functional feature is mapped to a conceptual subsystem in the conceptual architecture, and each function at the function level can be mapped to an operation of a class in the class diagram. Figure3, Figure4 and Figure5 illustrate the three views of conceptual architecture in different levels of details, among which the lower-level view is a refinement of the higher-level view.

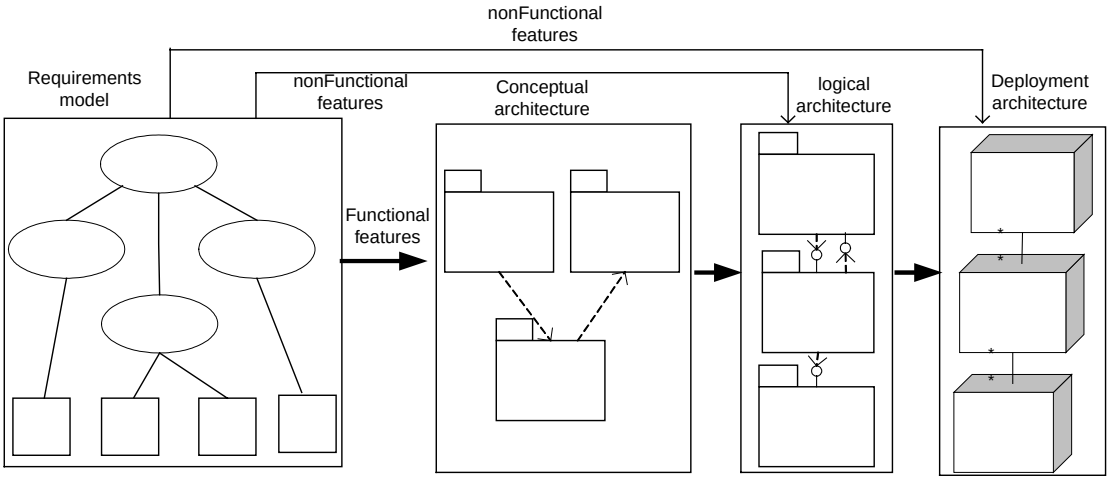

Figure 2. Mapping feature model to architecture models

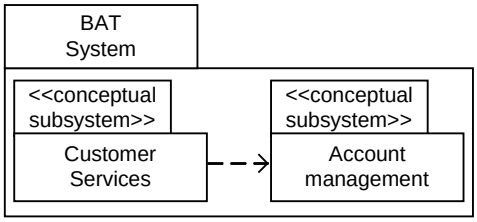

Figure 3. Business view of BAT conceptual architecture

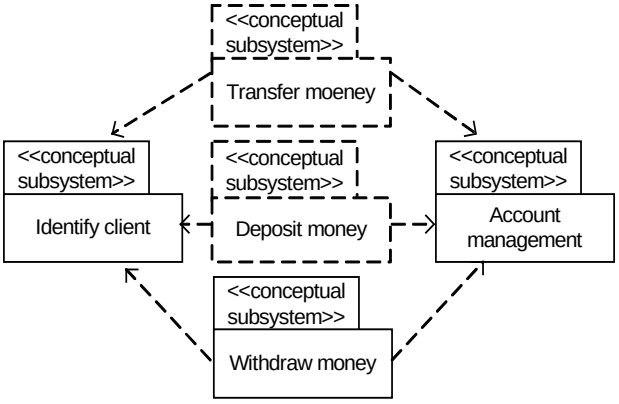

Figure 4. User view of BAT conceptual architecture

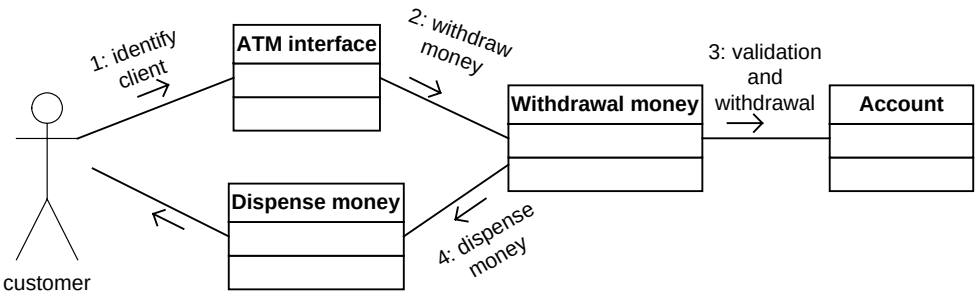

Figure 5. "Withdraw money" conceptual subsystem specification

Logical architecture

Logical architecture is the detailed architecture specification that defines the system components and interactions between components. Comparing with conceptual architecture, the logical architecture introduces more logical structures or mechanisms considering the implementation context and nonfunctional features. The form of the conceptual architecture may be adapted or even transformed. As shown in figure 6, considering nonfunctional feature "various UI", we apply "separation of concerns" strategy to the conceptual architecture. We separate responsibility for user interface from responsibility for transaction management. So we got a new system partition: "ATM interface" subsystem, "Transaction management" subsystem and "Account management" subsystem.

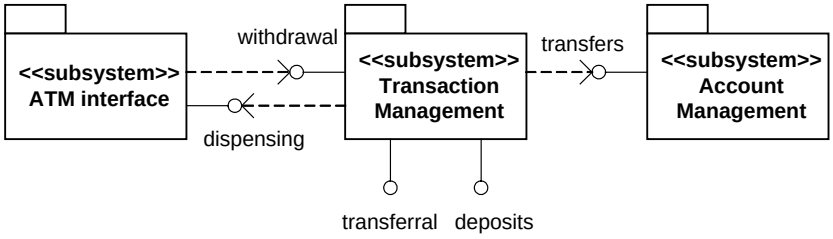

Figure 6. Logical architecture of BAT system

Deployment architecture

The deployment architecture focuses on how functionality is distributed among computational nodes and how computational nodes interact, to meet related nonfunctional features. As shown in figure 7, considering "remote client" and "fail-safe" features, a Three-Tier architecture style is selected, and the CHAP acknowledgement protocol is adopted to ensure connection safety. Some components identified in conceptual architecture and logical architecture, such as "withdraw money" subsystem, is distributed to the three nodes in this view.

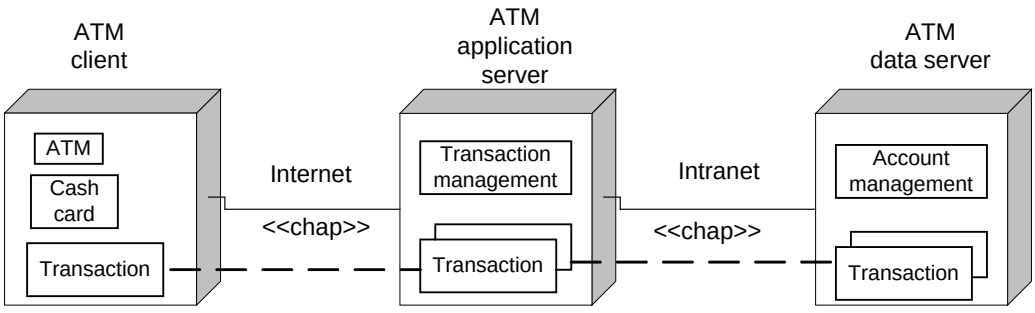

Figure 7. Deployment architecture of BAT system

5. Conclusion

In this paper, we analyze the gap between requirements and SA and the inadequacy of mapping approaches in traditional structured method and OO method. Based on this analysis, we propose a feature-oriented mapping and transformation approach. Our solution is to take feature-oriented as a paradigm both in requirements engineering and software architecting, so as to maintain direct and natural mapping between requirements specification and SA. Further, considering the different effect of functional features and nonfunctional features on SA, we address them separately in different models, iteratively and incrementally. So our approach can fit into iterative, incremental or evolutionary development paradigm.

We believe that feature-oriented development paradigm will gain more and more wide-spread acceptance. Further work will be to integrate our approach with existing CBSD development paradigm in order to support components reuse at different stages in software life cycle.

Acknowledgements

This effort gets support from China Natural Science Funds project under the contract 60125206. We would like to thank all the teachers and students who have discussed with us about this topic and give us good advice.

Reference

[1] Davis, A.M. The design of a family of application-oriented requirements languages. Computer 15 (5) (1982) 21-28.

[2] Kang, Kyo C. etc. Feature-Oriented Domain Analysis Feasibility Study (CMU/SEI-90-TR-21, ADA235785), CMU-SEI, 1990.

[3] Kyo C. Kang , Sajoong Kim etc. FORM: A feature-oriented reuse method with domain-specific reference architectures, Annals of Software Engineering 5 (1998) 143–168

[4] C. Reid Turner e.tc, A conceptual basis for feature engineering, The Journal of Systems and Software 49 (1999) 3-15

[5] Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology (IEEE Standard 610.12-1990). New York, N.Y.: Institute of Electrical and Electronics Engineers, 1990

[6] David Garlan and Mary Shaw, An Introduction to Software Architecture, In Advances in Software Engineering and Knowledge Engineering, Volume 1, World Scientific Publishing Company, 1993.

[7] Dewayne E. Perry and Alexander L. Wolf, Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, 17(4), 1992.

[8] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry Boehm, On the Definition of Software System Architecture, ICSE 17 Software Architecture Workshop, 1995.

[9] Ivar Jacobson, Grady Booch, James Rumbaugh. The Unified Software Development Process, Addison Wesley Longmon, 1999

[10] Merlin Dorfman. Requirements Engineering, SEI Interactive, March,1999

[11] Institute of Electrical and Electronics Engineers. IEEE Recommended Practice for Software Requiements Specifications (IEEE Std 830-1998), New York, N.Y.: Institute of Electrical and Electronics Engineers, 1998

[12] Dean Leffingwell, Managing Software Requirements: A Unified Approach, AT&T, 2000

[13] Karl E. Wiegers, Software Requirements, Microsoft Corporation, 2000