Architecture-Level Requirements Specification

Davor Svetinovic
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
dsvetinovic@uwaterloo.ca

Abstract

The large gap in the levels at which requirements are specified results in inadequate means for ensuring that business goals are properly supported. Architecture-level requirements specifications help us reduce this problem by providing necessary constructs and traceability mechanisms. Enhancing traditional requirements engineering approaches by incorporating architecture-level requirement specifications will facilitate business goals satisfaction and simplify the design of appropriate software architectures.

1. Introduction

Since its early days, software development has been implementation driven. Programming, still considered by many as the most important and difficult development activity, has attracted most of the research attention over time. While sufficient in some cases, programming has become a relatively routine activity compared to the other development activities in the development of today's large, complex, and constantly changing software systems. The main difficulty in today's development is not anymore *how* to build the system, but *what* to build and how to make it as adaptable to future change as possible [6].

Because of its early importance, implementation technologies and paradigms have influenced all development stages, even the early ones such as requirements analysis and design. For example, structured and object-oriented programming paradigms resulted in structured [12, 31, 30] and object-oriented analysis and design techniques [21, 9, 4]. This tradition continues with emergence of new methodologies such as aspect-oriented analysis [1], which has its origin in aspect-oriented programming paradigm.

The success of such approaches was mostly due to the fact that the traditional way of development focused on one product at a time [19]. A clear product-level requirements specification combined with low-level design, using structured or object-oriented concepts, was appropriate for a product development in relatively stable and well-understood problem domains.

Domain-level requirements analysis and specification appeared as a solution to a need for building software systems for large, difficult to understand, and changing problem domains. Goal-driven requirements engineering emerged as a leading approach for dealing with domain-level requirements for large systems [10, 23, 29]. The main emphasis of this approach was on making sure that software actually fulfills business goals. This goal fulfillment problem was one of the main weaknesses of the traditional product-level requirements engineering approach.

Now, with the emergence of new economic trends, the Internet as a business medium, software as a commodity, *etc*., even small systems have become much more difficult to build and maintain. New software paradigms and technologies such as web services, agility, and product lines, emerged to solve this new wave of problems. In this new situation, both business systems and software systems change faster than ever before. Naturally, both domain and product-level requirements specifications become obsolete very quickly, in some cases even before the product is built [17, 19].

2. Agility, Web Services, and Product Lines

In this section I would like to emphasize the commonalities of agile development paradigms, web services, and product lines as related to requirements specification. Even though all three concepts seem to have contradictory goals, they do share and contribute many new common development principles.

First, they deemphasize product-level requirements specifications. The agile development philosophy states that a detailed up-front specification of the product level requirements is unnecessary [18, 3]. Rather, agile followers believe that product-level requirements are best discovered on the fly, *i.e.*, by developers who directly communi-

cate with clients and implement these features without documenting them or preserving their rationale. Web services elevate responsibility for product-level requirements from the application developer to the service provider. Application developers can choose and integrate many different services which vary in particular feature details. A product line stresses the development of several product at a time. The main focus of the product line development is on the establishment of a robust architecture that will support many kinds of different product-level variations. In all three cases, we have a shift from product-level thinking to a whole new way of thinking, which allows us to think in terms of future changes and how we can accommodate them as easily as possible.

Second, all three approaches take into account the constant change in the problem domain. This is similar to the changes at the product-level.

So in summary, now we have the situation that software systems have to adapt to constantly changing problem domains, and also to be easily adaptable to new problem domains. This leads us to the following main constraints that requirement specifications should satisfy:

- Requirements artifacts should be reusable and easily modified, *i.e.*, they should be *assets*.
- The focus should be on the clear separation between commonalities and variabilities, from the requirements perspective.
- Domain and product-level requirements should be secondary as they describe mostly variations in functionality, and low-level requirements should be left to developers.
- We should emphasize stable, change resistent requirements with a high architectural impact.

3. Requirement Abstraction Levels

Requirements are specified either directly or indirectly for many different purposes, and as part of many different engineering activities. For our purposes, we can sort them according to different levels at which they usually appear:

1. Business-level requirements specification: Business-level requirements are most often indirectly specified as the part of business reengineering activities [17, 11, 2, 27]. The most common concepts that appear at this specification level are business goals, processes, resources, and rules. It can be argued that this is probably the most important type of requirements specifications, as the goal of software systems is to ultimately satisfy and contribute to the fulfilment of these business processes, goals, *etc*.

2. Domain-level requirements specification: As mentioned previously, domain-level requirements are one of the traditional approaches to the requirements specification [10]. Newer, more systematic versions of domain-level requirements engineering have received a lot of attention recently [5, 7, 24]. Most of its applications are in the area of business systems, which are getting increasingly complex and difficult to adequately support by software systems [8, 16, 15]. The most common concepts that appear at this specification level are user goals, user tasks, domain input, domain output, *etc*. More recent trend is the incorporation of agent-based analysis as the part of domain modelling [26, 20, 25, 14].

3. Product-level requirements specification: Product-level requirement specifications are the most common type of requirement specifications. There exists an extensive body of knowledge about them, and most of the previous research focused on perfecting different techniques used to elicitate, specify, and validate this type of requirements. The most common artifacts and concepts that occur as the part of product-level specifications are features, use-cases, functional lists, data input, data output, *etc*.

4. Design-level requirements specification: This is another type of well understood and widely used type of specification. A lot of efforts were invested into its standardization through Unified Modelling Language (UML) [21, 13]. UML artifacts present the most common types of concepts and techniques used to capture requirements at this level. This level acts as a transition phase between product-level specification and code-level requirement specifications.

5. Code-level requirements specification: Lastly, usually considered as a part of programming activity, low-level algorithm and data structure specification makes what we refer to as the code-level requirements specification. This is the type of the specification which most programmers are familiar with, as it is inseparable part of coding. It focuses mostly on the implementation related issues and constraints. This is also probably the best understood requirements specification level.

From this discussion, we can observe that most of the current forms of requirement specifications focus on the specification of functionality at the different levels. This leads us to the definition of the problem I am aiming to solve.

4. Problem Statement

While new development technologies and paradigms stress structure and quality over changeable functionality, traditional requirements engineering techniques still focus on primarily capturing the low-level functionality of the system. Structure and quality requirements are often deemphasized and hidden within specifications. The requirement engineering artifacts must be adapted to support this new development reality and improve the return on investment in all possible ways. Therefore, the problem that we are dealing with is: *How should we organize and specify requirements in such way to emphasize structure, quality, and stable requirements, and at the same time provide a way for capturing changeable and variable requirements?*

In addition, as change is occurring in both, business system and supporting software system, we have to perform the analysis and specify structural and quality requirements of both systems. A software system has to be adaptable to support also several different business systems, and to allow the evolution of all of them.

In my opinion, the most promising way to deal with these issues at design, implementation, and maintenance stages of software development cycle is the effective use of software architecture principles and techniques. Nevertheless, the effectiveness of software architecture techniques, especially when one has to develop multiple architectures at the same time, is in my opinion limited, as they are based on requirement specifications which are tailored to emphasize different issues such as low-level functionality, one product focus, *etc*. The goal is to try to solve this problem by introduction of architecture-level requirements specifications.

5. Proposed Solution

The hypothesis is that architecture-level requirements specification provides more support for the development of software systems using web services, agility principles, and product lines, than traditional domain and product level-requirements specifications. This support reflects through an improved architecture for the system, clear identification of common structural elements and functionality, and identification of variation points and constraints on the future evolution of the system. Also, architecture-level specification lies conceptually between domain and product-level specifications, allowing clear definition and verification of the mechanisms through which product features help achieve the business goals. Providing this traceability is identified as one of the most important requirement engineering problems [22].

Therefore, my work will focus on the definition of different requirement specification levels, together with the analysis and adaptation of different requirement specification techniques and artifacts to these levels. In particular, I will define a set of techniques and artifacts that can be used to capture architecture-level requirements. These include an architecture-level requirements specification method, which is based on the the focus shift from product and domain to their architectural properties, integration, and qualities.

6. Architecture-Level Use Cases

One of the already identified uses of architecture-level requirement specifications is the architecture recovery of software systems [28]. I successfully performed extraction and specification of architecture-level requirements in the form of *architecture-level use cases*.

Use cases — that is, narrative descriptions of domain processes — appear in different forms in all phases of a development cycle. They are typically used as the artifacts around which development cycles are organized. When used this way, all other activities and artifacts depend on them.

A use case describes the interactions between actors and system processes. A use case encapsulates responsibilities that are performed during a computation by actors and by system processes.

Architecture-level use cases are use cases that describe *logical processes* within the system. In my study, these use cases were not created by developers, but were generated using high-level responsibilities that were written as the part of the code documentation. The purpose of this generation was to document dynamic processes within the system. This was a technique used as the part of the logical architecture view in order to present dynamic interactions in a comprehensible format.

Requirements were discovered and abstracted from the method-level to the subsystem level. While module-level responsibilities provide a compact way to encapsulate and represent architecturally significant features, method-level responsibilities are used to understand and present module and subsystem interactions using architecture-level use cases. The advantage of architecture-level use cases over other presentations like sequence diagrams is that they present dynamic aspects in a comprehensible way while hiding low-level details.

Architecture-level use cases were built using navigational capabilities of several code-browsing tools in conjunction with documented responsibilities. The main value of this approach was not in a documentation of all possible use cases, but in an ability to recover them as needed. Although responsibilities were not required to be documented within source code, the advantage of having them documented there is that a transition from architectural level analysis to low-level design analysis is seamless. Bellow is an example of a fully developed architecture-level use case:

- *Name:* Play Song
- *Actors:* End-user
- *Stakeholders:*
  1. End-user
  2. Music provider
- *Event:* User pushes play button
- *System:* xmms, libxmms, input, output, visualization
- *Purpose:* Describe collaboration among subsystems to accomplish "play" functionality
- *Priority:* 10/10-core business process, crucial for business operation
- *Overview:* Input stream is processed to produce wanted output (song playing or streaming to a file on hard disk)
- *References:* None
- *Related Use-Cases:* Setup
- *Responsibility:* Play media stream or write it to a file
- *Preconditions:* Play-list was configured, Setup use-case successfully performed
- *Postconditions:* System stops playing, after input stream end, if Repeat option is turned off.
- *Invariants:* None
- *Main Scenario:*
  1. xmms: User interface component signals "play" event is raised.
  2. xmms: Signal to input subsystem to start processing data
  3. xmms: Connector between input and output is established
  4. input, output, visualization: Start processing data streams
  5. input: If end of the stream signal xmms and stop
  6. xmms: If "Repeat" option turned on signal input to start processing again, else "stop" signal to output and visualization
- *Alternatives:*
  - step 4: data stream disconnected before end of it (file deleted, network connection went down, etc.) — raise exception and inform user
  - step 4: special effects events raised — activate appropriate plugin, which alters output
- *Quality Attributes:*
  1. Responsiveness — events are handled without delays
  2. Reliability — user is informed within 1 second if system stops due to data stream problems
- *Technology:* Network access support for network streams
- *Special Requirements:* For low-end systems, output processes have higher priority over visualization and affect subsystem's processes
- *Open Issues:* None

A single column format was used to document this particular use case. One could also use multiple column format to emphasize subsystems and modules. The second option has a drawback that it is harder to format text properly thus increasing production and maintenance time.

7. Current Work and Open Questions

Currently, I am involved in an exploration of the following topics:

- Identification of architecture-level requirements: properties and patterns. This topic involves analysis of which properties of the requirements have a significant architectural impact. This knowledge can be used to discover them and isolate from the different software requirement specification documents.
- Recovery of the architecture-level requirements from code, UI, and deployment properties. This recovery is a process of abstracting and combining requirements all the way up to the business level. Its value is in being able to analyze how software impacts the business. This analysis is important in situations in which new software is acquired and business is tailored to it.
- Analysis of the architecture-level requirements change. This analysis is an observational study of several systems to try to discover the evolution patterns and properties of the requirements that actually change over time.
- Techniques for the architecture-level requirements specifications. There are two main techniques:

– Proposal of a new way to organize software requirement specifications. The aim is to structure the requirement specifications in order to preserve and emphasize business and software architecture requirements and concepts.

– Architecture-level use cases for capturing dynamic properties and functional requirements at the subsystem level.

8. Conclusion

This paper has attempted to emphasize the importance of the conceptual shift from the traditional domain and product-level requirement specifications to multiple level specifications and to architecture-level requirements specifications, in particular. The main purpose of this shift is to accommodate the development using new software paradigms. Also presented were some parts of the work that was done as a part of a study in software architecture recovery.

References

[1] *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, Mar. 2002.

[2] W. Aalst and K. Hee. Framework for business process redesign. In J. R. Callahan, editor, *Proceedings of the Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 95)*, pages 36–45, Berkeley Springs, 1995. IEEE Computer Society Press.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, Massachusetts, first edition, 1999.

[4] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Boston, Massachusetts, second edition, 1994.

[5] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modeling early requirements in tropos: A transformation based approach. In *AOSE*, pages 151–168, 2001.

[6] F. J. Brooks. No silver bullet. *Computer*, 20(4):10–19, April 1987.

[7] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. *Lecture Notes in Computer Science*, 2068:108–??, 2001.

[8] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: The tropos project. *To Appear in Information Systems, Elsevier, Amsterdam, The Netherlands*, 2002.

[9] P. Coad and E. Yourdon. *Object Oriented Analysis*. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[10] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[11] T. H. Davenport. *Process Innovation – Reengineering Work through Information Technology*. Harvard Business School Press, Boston, first edition, 1993.

[12] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, first edition, 1978.

[13] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, Massachusetts, second edition, 1999.

[14] P. Giorgini, A. Perini, J. Mylopoulos, F. Giunchiglia, and P. Bresciani. Agent-oriented software development: A case study. *In Proc. of the 13th Int. Conference on Software Engineering & Knowledge Engineering (SEKE01), Buenos Aires, Argentina*, 2001.

[15] F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos software development methodology. *Technical Report No. 0111-20, ITC - IRST. Submitted to AAMAS '02. A Knowledge Level Software Engineering 15*, 2001.

[16] F. Giunchiglia, A. Perini, and F. Sannicolo. Knowledge level software engineering. *In Springer Verlag, Editor, In Proceedings of ATAL 2001, Seattle, USA. Also IRST TR 011222, Istituto Trentino Di Cultura, Trento, Italy*, 2001.

[17] M. Hammer and J. Champy. *Reengineering the Corporation: a Manifesto for Business Revolution*. Nicholas Brealey P., London, first edition, 1995.

[18] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, Boston, Massachusetts, first edition, 2000.

[19] P. Knauber, D. Muthig, K. Schmid, and T. Wide. Applying product line concepts in small and medium-sized companies. *IEEE Software , Vol.17, Iss.5*, pages 88–95, 2000.

[20] M. Kolp, P. Giorgini, and J. Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In J.-J. Meyer and M. Tambe, editors, *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, pages 146–158, 2001.

[21] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Englewood Cliffs, N.J., second edition, 2001.

[22] S. Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, Boston, Massachusetts, first edition, 2002.

[23] J. Mylopoulos, L. Cheung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of ACM, Vol. 42, No. 1*, 1999.

[24] J. Mylopoulos, L. Chung, S. Liao, H. Wang, and E. Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, /2001.

[25] J. Mylopoulos, M. Kolp, and J. Castro. UML for agent-oriented software development: The tropos proposal. *Lecture Notes in Computer Science*, 2185:422–??, 2001.

[26] J. Mylopoulos, M. Kolp, and P. Giorgini. Agent-oriented software development. In *SETN*, pages 3–17, 2002.

[27] S. R. Schach. *Classical and Object-Oriented Software Engineering With Uml and Java*. McGraw-Hill, fourth edition, 1998.

[28] D. Svetinovic. Agile architecture recovery. Master's thesis, School of Computer Science, University of Waterloo, 2002.

[29] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Vol. 26, No. 10*, 2000.

[30] E. Yourdon. *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1988.

[31] E. Yourdon and L. Constantine. *Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1979.