Why Use Case Driven Approach?

JULIJANA LINIĆ
Department of Software Development
Financial Agency
Vranovina 30, 10000 ZAGREB
CROATIA

Abstract: - Inadequate requirements specifications not understandable to users are one of the predominant causes of failure in the development of software systems today. The aim is to find a technique understandable to users in order for them to be able to validate these requirements and verify whether these requirements are really what they need. Further, this manner of presenting requirements must be familiar to developers and must facilitate development of software systems. On the other hand, the aim is finding the way of specifying requirements that would be in accordance with modern techniques of developing software systems such as traceability and iterative and incremental development. Use cases are a reliable technique of resolving the problems mentioned.

Key-Words: - Use cases, Use-case model, Requirements, Requirements specification, Requirements techniques, Traceability, Iterative, Software development, Methodologies

1 Introduction

The term use case was introduced in software engineering by Ivar Jacobson. In fact, Jacobson is often thought of as being the father of the use case and the father of Unified Process (UP, USDP) - one of the most widespread world's methodologies for developing software systems. According to his paper presented in OOPSLA '87' "a use case is a special sequence of transactions performed by a user and a system in a dialogue". This is quite similar to our current (informal) definition. He developed a separate model for describing an outside perspective of a system and he called it a use-case model. By 'outside', he meant a black-box view of the system. The internal structure of the system was of no interest in this model. The use-case model was a model of the functional requirements of the system. In 1994 he added the requirement that a use case must give a "measurable value" to a "particular actor". Over the years the use case has matured. Jacobson included use cases as part of an overall system development lifecycle methodology called Objectory, which he marketed as a product and built a company around. Later, Jacobson's Objectory and his methodology were purchased by Rational Software, and in 1997 use cases became part of the Unified Modeling Language (UML) – a general-purpose visual modeling language for systems. At the same time, Objectory became part of the Unified Process. Today, Unified Process is developed as a commercial methodology called Rational Unified Process (RUP) and since 2003 its owner has been IBM. Rational Unified Process is today one of the well-known world's commercial methodologies for the development of software systems.

In UML the use cases are diagrams. But it is important to say that the use cases are primarily textual documents in form of specification which contains the description of the use case. There are many use-case formats in use in various projects. There are also many different popular styles of use case. Generally, a use-case specification may have a formal and an informal form. However, each use-case description contains a brief description, a flow of control, a base flow and alternative flows, subflows (reusable at many places within the same use-case description), preconditions and postconditions.

Use cases are created for expressing functional requirements of a system, but they are more than a requirements technique. Use cases are traceable to analysis, to design, to implementation and to test. For each use case in the use-case model we create the collaboration (a view of participating classes) in analysis and design. Each use case results in a set of test cases. Use cases are important for designing user interfaces and for structuring the user manual. Use cases also perfectly match business processes. This approach to software development, which starts with identifying all use cases and specifying each one of them in requirements, analyzing and designing each one of them in analysis and design respectively, and finally testing each and every one of them in test, is called use-case driven development.

2 Problem Formulation

Many studies show that one of the primary challenges of vital importance to any information system development is the ability to elicit the correct and necessary system requirements and specify them in a manner that is understandable to users in order for those requirements to be verified and validated. The information technology community has always had problems trying to specify requirements, especially functional requirements, to users. There were tendencies to produce diagrams and specifications that were loaded with terminology and notation resembling a computer code. The traditional ways of expressing functionality to users are requirements specifications, functional decompositions, data-flow diagrams (DFDs), entity-relationship diagrams (ERDs), prototypes, etc. These modes of expressing functionality of a system are not understandable to users.

Traditionally, requirements are specified in lists and expressed in terms of ‘’the system shall’’. The requirements lists provide a comprehensive catalog of every function that the system should perform. In most cases these lists contain duplicate or conflicting requirements.

Another attempt to describe functionality of the system is functional decomposition. This method takes the major function of the system, the highest level function and breaks it down to subprocesses, and sub-subprocesses, and so on. When the processes are small enough, they become a program. Functional decomposition is a remnant of the older analysis and design approaches. It is tightly linked to structured systems development, meaning COBOL and mainframes. It is not usable for an application that is Web-based or object-oriented.

The world-famous methodologies, like structured techniques and information engineering, have a traditional approach to software development. Their main artifacts for developing systems are data-flow diagrams and entity-relationship diagrams. In the design phase data-flow diagrams are sometimes called flowcharts. These methodologies are similar because there exist two parallel separated ways of developing software system. These are the process model and the data model. The main difference between the two types of methodologies is that the structured techniques are based on processes, i.e. the process model is their primary artifact, while information engineering is based on data, and in this approach the data model is the primary artifact. Data flow diagrams help to show a system as a set of groups of interacting processes. They represent the dynamic view of the system and focus on what happens inside the system. The data flows from one process to another, and then stops in a data store. Entity-relationship diagrams show how the data is stored in an application. They show details of entities, attributes, and relationships. Also, this diagram is used to present a logical data model and dictate the structure of a relational database. Both of the artifacts can be useful in the design phase primarily for non-objected software development, but they are not requirements artifacts and they are confusing for users.

Prototypes were long considered as a good requirement tool. They give users a realistic demonstration of what a system will be able to do when it is completed. But when using prototypes, users are concentrating on the details of user interface and not on the requirements of the system. Prototypes also encourage users to think that the prototype is the system. This approach leads to the misperception and quick-and-dirty coding because users are impatient to develop a real system.

Therefore, there is a need for a different technique to describe the user’s requirements. This new technique must allow users to verify whether these requirements are really what they need.

3 Problem Solution

The use-case modeling is one of the techniques that resolve the problems with requirements mentioned above. The strength of this modeling is that it facilitates usage-centered development. The usage-centered approach is a new approach in software development. It requires concentrating on users’ needs and on the reasons why the system should be developed in order to successfully plan, analyze, design, construct and deploy an information system.

3.1 Functional and Nonfunctional requirements

There are two basic categories of requirements: functional and nonfunctional requirements.

Functional requirements are those actions that a system must be able to perform, without taking physical constraints into consideration. The functional requirements specify the input and output behavior of a system. Nonfunctional requirements specify other qualities that the system must have, such as those related to the usability, reliability, performance, and supportability of the system [3].

Use cases describe functionality of the system. This technique cannot be used to describe nonfunctional requirements. For describing such requirements there are the supplementary specifications that are delivered together with a use-case model.

3.2 The Basic Concepts of Use-Case modeling

A use case specifies sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors [9].

Use cases are presented formally or informally in text form or in a graphically high-level view by UML use case diagrams.

We often start modeling with a use case diagram. The use case diagram shows three aspects of the system – actors, use cases and the system boundary.

In UML 2, the system boundary is referred to as the subject. The subject is defined by who or what uses the system (i.e., the actors) and what specific benefits the system offers to those actors (i.e., use cases). The subject is drawn as a box with the actors presented outside this box and the use cases inside. Actors represent the roles that some external entities (people, other systems or devices) take on when communicating with the particular use cases in the system. Actors are drawn as a stick figure or as a class icon stereotyped «actor». A use case describes the behavior that the system exhibits to benefit one or more actors. A good way to find use cases is by asking the questions like: "How does each actor use the system?" and "What does the system do for each actor?"

The use case diagram helps to identify use cases. However, use cases must be specified in the textual form. The document which specifies use cases is known as a use case specification. There are many forms of use case specifications. A template for this specification must be known within each organization or project.

The simple template for a use case specification generally contains:

1. a use case name
2. a unique identifier
3. a brief description
4. actors (there are primary actors who trigger the use case and the secondary actors who interact with the use cases after it has been triggered)
5. preconditions (system constraints that affect the execution of a use case)
6. postconditions (system constrains arising out of the execution of a use case)
7. main flow (the sequence of declarative, time-ordered steps in the use case)
8. alternative flows (the list of alternatives to the main flow which we can present within a use case or as a separate use case)

Apart from these a template may also include some extra information.

3.3 Advanced Use-Case Modeling

There are several advanced techniques for use-case modeling:

1. actor generalization
2. use case generalization
3. «include» relationship
4. «extend» relationship

Actor generalization allows to factor out into a parent actor behavior that is common to two or more actors. There is a substitutability principle – a child actor may be substituted anywhere a parent actor is expected. The parent actor often specifies an abstract role, i.e. it is an abstract actor.

The use case generalization allows to factor out features that are common to two or more use cases into a parent use case. The child use cases inherit all the features of their parent use case. Also, the child use cases may add new features or they may override parent features.

The relationship «include» allows to factor out steps repeated in several use case flows into a separate use case. The including use case is the base use case and the included use case is the inclusion use case. The base use case is not complete without its inclusion use cases and it cannot exist without them. An inclusion use case may exist without its base use case - it may be complete or incomplete.

The relationship «extend» adds a new behavior for the existing use case. There is a principle similar to the one in the relationship «include». The base use case has extension points which may occur between the steps in the flow of events. The difference between the «include» and the «extend» is that the base use case is complete without its extension use cases and it does not know anything about possible insertion segments, it just provides hooks for them. Therefore, the extension use cases are added to an overlay on top of the flow of events. The extension use case is generally not complete. It may also be a complete use case, but this is rare.

3.4 The Rules of Writing Good Use Cases

The main rules of writing use cases are:

1. keep use cases short and simple – include only enough detail to capture the requirements
2. focus on the *what*, not the *how* – when writing use case we must concentrate on *what* the actors need the system to do, and not on *how* the system should do it
3. avoid functional decomposition – the use case model is not used for structuring requirements and we cannot create a set of "high level" use cases and then break these down into a set of lower-level use cases and so on, until we end up with "primitive" use cases
4. consider not using advanced techniques - use them only where they simplify the use case model

3.5 The Styles and Formats of Use Cases

Use cases are a very flexible and extendable software development technique. There are many styles and formats of use cases.

Generally, there exist two basic flavors of use case models: essential use case models and system use case models. An essential use case model is technology-independent view of behavioral requirements. It is also known as a task case model or an abstract use case model. An approach to software development with the essential use case is presented in the book *Software for Use* (Constantine and Lockwood 1999). Essential modeling is a fundamental aspect of usage-centered designs and it captures the essence of problems through technology-free, idealized, and abstract descriptions. A system use case model describes in detail how users will work with the system, including references to user-interface aspects. However, a system use case considers technical considerations, and it is also known as a concrete use case model or a detailed use case model. System use cases are the primary requirements artifact for the Rational Unified Process (RUP).

Regardless of this classification, use cases may describe requirements in a formal or an informal way. In the formal approach, the use case model is large, comprehensive and detailed. In the informal approach, the use case model is high level and omits cursory details. There were projects which successfully implemented systems using each one of these approaches. Also, there were projects which failed with each of these approaches. The basic difference between these successful and unsuccessful projects was the planning. The developers of a successful project understand the strengths and the weaknesses of each approach and plan accordingly. If a large project creates a high-level, informal use case model, requirements will be missing, incomplete and ambiguous. On the other hand, very detailed, comprehensive model with lots of use cases, and with lots of logic and business rules in the flow of events can be unsuccessful because such amount of details creates the danger of losing the thread of the development.

There are also many different use-case formats in use in various different projects and texts:

1. brief descriptions – a short paragraph that describes something that the system does
2. outlines – a numbered or bulleted list of events and responses
3. table formats – form of table of actors' actions and system's responses
4. "black box" – a view focusing on the actions taken by the actor and the system's response
5. structured English forms – sequential paragraphs of the text or narrative form

Different use-case formats are often associated with different points in the evolution of a use-case model. We often say that use cases have a complex life cycle. They mature through a number of development stages, from discovery to implementation, and eventually to user acceptance. The life cycle of the use case continues beyond its authoring to cover activities such as analysis, design, implementation, and testing. We say that the process employs a "use-case driven approach" e.g. the uses cases defined for a system are the basis of the entire development process. The use-case model is the result of the requirements discipline. Use cases are realized in analysis and design models. They are implemented in terms of design classes. When the code has been written, it enables a use case to be executed; now, the use case is in the implemented state. Also, the use cases are the basis for identifying test cases and test procedures. In this phase of software development, the system is verified by performing each use case. When the use cases pass an independent user-acceptance testing, the system is in the accepted state. Use cases can also help with the supporting disciplines. In the project management discipline use cases are used as a basis for planning and tracking the progress of the development system. This has the primary affect on iterative development where use cases are the basic planning mechanism. In the deployment discipline use cases represent the base for writing user's manuals.

3.6 The Main Properties of Use Cases Significant for Developing Modern Software Systems

The main properties of use cases as a requirements artifact for developing software system are their traceability and the ability of using them for iterative and incremental development of the software system.

3.6.1 Traceability

Experience has shown that the ability to trace requirements artifact through the stages of specification, architecture, design, implementation, and testing is a significant factor in assuring a quality software implementation. As mentioned in the previous section, use cases encourage traceability. The use case model may be traced to the use case realization, and the use case realization to classes (code) that implement the collaboration. On the other hand, use cases may be traced to test cases. However, each use case has a variety of possible scenarios that can be tested, and that is the first step of traceability from a use case to test cases. Another step of this traceability is trace from scenarios to test cases. Also, in an early stage of software development system stakeholder needs are traced to product features, and product features. These product features are then traced either to use cases or to supplemental requirements, which depends on whether the requirements are functional or nonfunctional. This traceability is shown in Fig. 1.

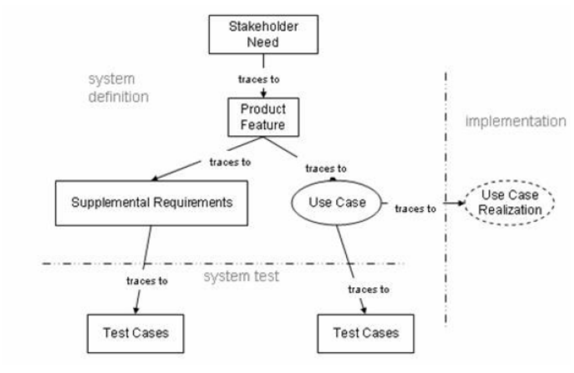

Fig.1: Generalized traceability model

Modern modeling tool such as RequisitePro and DOORS enable describing traceability. For assuring traceability, there exists traceability matrix similar to the one shown in Table 1. The requirements matrix may be manually created, but it is not recommended.

Table 1: Traceability Matrix – System Features to Use Cases

	Use Case 1	Use Case 2	…	Use Case n
Feature #1	X			X
Feature #2		X		X
…			X	
Feature #m		X		X

3.6.2 Iterative and Incremental Development

Incremental development is the development of a system in a series of versions, or increments. A subset of functionality is selected, designed, developed, and then implemented. Additional increments build on this functionality until the system is completely developed. Iterative development is planned rework of existing functionality. It is common practice to use the term iterative development to represent both concepts [4].

In the early increments the use cases that contain basic functionalities for users as well as architecturally significant use cases are selected. At the end of each increment each of these selected use cases is modified according to the lessons learned in the increment. However, use cases then become primary planning mechanism for iterative development.

4 Conclusion

This article is an attempt to show that traditional techniques of gathering the requirements are not acceptable to users. They cannot verify whether the specified requirements meet their needs. On the other hand, the gathering and specifying requirements is the key phase in the development of software systems and many projects were unsuccessful because there were omissions in this phase. There is a need for a technique that would be familiar to users on the one hand and to analysts, designers, implementers and testers on the other. However, this technique must provide requirements traceability. In addition to that, it must provide iterative and incremental development because the waterfall approach is not appropriate for the development of modern software systems, and most world-famous methodologists abandoned this method. The solution of this problem is use cases. The strength of this technique is their formal and informal format. It gives a possibility to use one of the well-known world's methodologies like a UP, e.g. RUP, or one of the agile methods for developing software system. Further, it gives a possibility of creating a new methodology more or less similar to these methodologies. The format of the use case can be chosen according to the needs of a project.

The weakness of the use case as a requirements technique is that the nonfunctional requirements cannot be presented with the use-case model. Therefore, for presenting overall requirements of the system, complement specifications, known as supplementary requirements, must be delivered together with the use-case model. However, that should not be a problem. The strength of use cases surpasses their weaknesses.

References:

[1] J. Arlow and I. Neustadt, *UML 2 and the Unified Process, Practical Object-Oriented Analysis and Design*, 2nd Edition, Addison Wesley, 2005.

[2] D. Kulak and E. Guiney, *Use Cases Requirements in Context,* 2nd Edition, Addison Wesley, 2003.

[3] F. Armour and G. Miller, *Advanced Use Case Modeling,* Addison Wesley, 2001.

[4] K. Bittner and I. Spence, *Use Case Modeling*, Addison Wesley, 2002.

[5] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Use Case Approach*, 2nd Edition, Addison Wesley, 2003.

[6] L. Whitten, L. D. Bentley, and K. C. Dittman, *System Analysis and Design Methods*, 6th Edition, McGraw-Hill, 2004.

[7] A. Cockburn, *Writing Effective Use Case,* Addison Wesley, 2001.

[8] "Rational Unified Process Documentation", IBM Corp, http://www-128.ibm.com/developerworks/rational/

[9] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual*, Addison Wesley, Reading – MA, 1999.

[10] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P.Krause, Rule-based generation of requirements traceability relations*, The Journal of Systems and Software*, No.72, 2004, pp. 105-127.

[11] S. S. Some, Supporting use case based requirements engineering*, Information and Software Technology*, No.48, 2006, pp. 43-58.

[12] M. Ratcliffe and D. Budgen, The application of use case definitions in system design specification, *Information and Software Technology*, No.43, 2001, pp. 365-386.

[13] I. Diaz, F. Losavio, A. Matteo, and O. Pastor, A specification pattern for use cases, *Information & Management*, No. 41, 2004, pp. 961-975.