Gathering Requirements
The Use Case Approach

by

Chaelynne M. Wolak
wolakcha@scsi.nova.edu

A paper submitted in fulfillment of the requirements
for DISS 725 – Research Paper Three
DISS 725 Spring 2001

School of Computer and Information Sciences
Nova Southeastern University

June 2001
Introduction

The Standish Group estimates that almost 80,000 projects were cancelled in 1995 alone. One of the primary reasons for these cancellations lies in creating understandable written communications (i.e. software specifications). This lack of understanding not only comes from the software development team but also the customer (Cobb, 1996).

Many companies have fallen into the trap of not defining system requirements at the beginning of the Software Development Life Cycle (SDLC). This was especially true during the 1970’s and early 1980’s when software development projects had resulted in significant costs (Smith, 2000).

Now, it is widely recognized that requirements gathering is vital to the SDLC process (Smith, 2000). This is because the quality of any software product depends on the quality of the raw materials that are fed into it. Thus, poor requirements lead to poor software (Wiegers, 1999).

One of the greatest challenges in software development is sharing the vision of the final product with the customer. When there are misunderstandings, the more likely “surprises” appear and in the software industry, “surprises” are not good news. Therefore, there needs to be a way to accurately capture, interpret, and represent the voice of the customer when specifying software requirements (Wiegers, 1997).

One solution is to gather requirements with Use Cases. This academic research paper begins with the background and definition of Use Cases. Next, reviews of the various Use Cases modeling approaches are detailed. In addition, two real life examples of Use Cases are outlined. Next, the common Use Cases pitfalls as well as effectively applying them are briefly described. Lastly, the paper concludes with a summary on gathering requirements with Use Cases.

Use Cases Background & Definition

For a long time, in both the object-oriented (OO) and traditional development world, people have used scenarios (Use Cases) to help them understand requirements. Nevertheless, they were treated very informally. The importance of Use Cases had not taken formal hold until 1994 (Fowler & Scott, 1997).

In 1994, Ivar Jacobson first introduced Use Cases as part of an OO development methodology in his book. It had become the primary element in project development and planning within the OO community (Fowler & Scott, 1997). It was not until the release of the Unified Modeling Language (UML) specification version 1.1 that the scope of Use Cases had broadened (Hurlbut, 1997). Recently, Larry Constantine and other researchers have extended the Use Case concept into a requirements gathering technique (Wiegers, 1997). Thus, the Use Case analysis is now used most often in the early stages of software development (Collins-Cope, 2000).
A Use Case (also called a scenario) captures a contract between the stakeholders of a software system about its behavior (Cockburn, 2000). Use Cases describe the software system’s behavior under various conditions as it responds to requests by the users. Essentially, it provides an easy-to-understand representation of the functional requirements for a software system (McGregor & Major, 2000).

There are many reasons why Use Cases should be used for requirements gathering. The reasons are as follows (Collins-Cope, 2000):

- They officially documents the development process
- They can separate the functional requirements from the business process
- They serve as requirements documentation for software development
- They are easy-to-read and easy-to-understand
- They can be used to categorize and rank requirements
- They can identify possible system/component reuse

Due to these benefits, Use Cases are now becoming customary as a method for gathering requirements (Wyder, 1996).

A Use Case collects the various scenarios that can happen within the software system. Use Cases are in text form, but are known to be diagrams, flow charts, sequence charts, Petri charts, or programming languages. Generally, though, simple text is the best form for Use Cases (Cockburn, 2000). A sample Use Case diagram and text form are shown in Appendix A. The basic structure of a Use Case includes the following elements: primary actor/actors, goals, main success scenario, and extension conditions (Fowler & Scott, 1997). Each of them is briefly described below.

**Primary Actor/Actors.**

The primary actor/actors are the users of the software system. They can be a person or another computer system. Essentially, though, they are external entities that will eventually interact with the system to achieve a desired goal (Cockburn, 1999).

**Goals.**

The goals represent specific functionality required of the software system (Cockburn, 1999). In some cases, goals can be separated into two categories: system interaction goals and user goals. System interaction goals reflect things the user is doing with the system. User goals are what the user wants to achieve from the system (Fowler & Scott, 1997).

**Main Success Scenario.**

The main success scenario describes a case in which nothing goes wrong. All goals are met with this scenario (Cockburn, 2000). It is the utopia situation.

**Extension Conditions.**

Extension conditions are also called failures. Upon completion of the main success scenario, all extension conditions that could occur should be listed. This is usually done during a brainstorming activity (Cockburn, 2000).
Use Case Models

As discussed previously, Use Cases are now broadened to cover requirements gathering. Use Cases can be used in many different situations. Thus, there are no two Use Cases alike (Cockburn, 2000). This section details at least three Use Case Models. They are the following: Adaptive Use Cases, Agent Based Use Cases, and Change Cases.

**Adaptive Use Cases.**

Adaptive Use Cases provide a behavior-oriented perspective through its focus on scenarios. In contrast, traditional Use Cases generalize collective behavior. Therefore, adaptive frames focus on package boundaries and variant scenarios than just the secondary actors and system boundaries (Hurlbut, 1997).

The adaptive frame requires an additional constraint to a Use Case by detailing the archetype scenario that has no conditional branching. The archetype scenario is the basic course of actions required to achieve the desired goal within the software system. Once this scenario is detailed, then all exceptional behavior and error handling conditions are described as other scenarios called variant scenarios (Hurlbut, 1997).

These variant scenarios are established based on the “adaptive” relationship to the archetype scenario with the results being placed in a set of adaptive frames. It is only the actions common between the scenarios that are reused. The integration of the adaptive frames associates a Use Case with its archetype scenario and incorporates all variant scenarios as an adaptation of its base line. Thus, each step in the Use Case maintains a formal link through this adaptive mechanism (Hurlbut, 1997).

**Agent Based Use Cases.**

In a traditional Use Case, all conditional or special conditions are detailed as “extensions.” Agent Based Use Cases expand this traditional Use Case approach to include depictions of how users and organizations enter and interact with each other. This is in addition to detailing the system interaction (Hurlbut, 1997).

Agents can be classified as weak or strong. A weak agent is one that is described as autonomous, social, reactive or proactive. A strong agent tends to add mentalistic notions such as beliefs, desires, rationality, veracity, and adaptability. Generally, agents are used when proactive behavior is required. Traditional Use Cases do not indicate which scenario has the most control or decision making behavior where Agent Based Use Cases do (Hurlbut, 1997).

**Change Cases.**

Change cases are used to capture the anticipated future requirements of a software system. They are used to increase the software system’s robustness by tracking all change requirements. The premise of Change Cases is that the evolution of the software system and maintenance will be much easier once the interface is correct (Hurlbut, 1997).
Four types of changes are considered. They include market demands, business requirement changes, legislative/regulatory change, and imaginative users. These changes are then characterized based on three factors. They are focus (i.e. system responsibilities), scope (i.e. artifacts), and degree of definition (i.e. extent of the change). Nevertheless, Change Cases trace change requests, identify the most important changes, and help determine which ones can be incorporated within the budget (Hurlbut, 1997).

Real-Life Examples of Use Cases

Due to the various adaptations of Use Cases, every company uses them differently. In this section, two real-life examples of Use Cases are briefly described. One is specifying the requirements for a chemical tracking system and the other is an electronic commerce quotation system.

Chemical Tracking System.

Eastman Kodak applied Use Cases to specify the requirements for a chemical tracking system. The team consisted of six chemists. Each of the chemists had to think of the reasons why they would need to use a chemical tracking system. Each of these reasons had become a Use Case (Wiegers, 1997).

Each Use Case was then explored in depth during a workshop. A flipchart was used to describe the Use Case. The purpose/goal of the Use Case was listed at the top of the flip chart. In addition, it was determined the user classes – those individuals that would be using the system during that specific scenario such as chemist or technician. The frequency of the Use Case was also determined and noted (Wiegers, 1997).

Next, the bulk of the workshop was spent detailing the actions of the users and the possible software system responses. These two items were placed in columns, side by side with sticky notes. Sticky notes had made it easier to move the various user actions/system responses around as other thoughts had created more actions and system responses. For example, an user action where an user enters the order number he/she wants to view is aligned with the system response of order is displayed with details shown (Wiegers, 1997).

After each workshop (i.e. each Use Case explored in detail), the flipchart with the sticky notes was used to extract requirements. These requirements were then used to create the structured software requirements specifications (SRS). The SRS was updated each week after reviews with the team. Nevertheless, it was determined that the Use Case strategy of consistently asking “What do you need to accomplish with the system?” kept everyone focused on visualizing how the software system ought to perform some function (Wiegers, 1997).

Electronic Commerce Quotation System.

An electrical product company whom had many plants throughout Europe wanted to improve the quotation process for product pricing and placing orders for customers. The existing process had consisted of each plant with their sales person using their own computer program providing the quotation. As a result, the function of providing quotations and
ordering products were inconsistent plant to plant. Therefore, it was determined that a more integrated system used among all the plants would provide the lowest prices and aid in streamlining the company’s bidding process (Anton, Dempster, & Siege, 2000).

It had become apparent that a company-wide electronic commerce quotation system was required. The company used a goal approach Use Case. Each goal of this quotation system had become one Use Case. Overall, there was a total of 52 Use Cases generated during the inquiry-driven analysis. This analysis had taken each goal and described its Use Case with the following information (Anton, Dempster, & Siege, 2000):

- A unique identifier
- A title
- An overview
- Pre and post conditions
- A main scenario
- Zero and more secondary scenarios
- The required graphical user interfaces (GUIs)
- List of Use Cases included or extended by each Use Case
- Revision history

The company had used Microsoft Excel to create the spreadsheet workbook for each Use Case. In addition, the company had wanted traceability as a way to track and document changes to each Use Case. Then the analysis team used these 52 Use Cases to create their SRS (Anton, Dempster, & Siege, 2000).

Use Case Pitfalls

Unfortunately, there are instances where some companies apply Use Cases ineffectively. Some of the common problems encountered are poor quality requirements and poor quality system design, along with a huge amount of wasted time and effort (Korson, 1998). Next is a review of some of the common Use Case pitfalls.

One of the most common pitfalls is generating too many Use Cases. Functional requirements, interface requirements, and business requirements should not be included in a Use Case. When all these requirements are placed into Use Cases, the number of Use Cases can become too overwhelming (Rosenberg & Scott, 2001). For example, one company generated 12,386 Use Cases for their software project. However, none of those Use Cases was useful because they did not contain the right information. Thus, over a half of million dollars was wasted (Korson, 1998).

Another pitfall is only describing system interactions and forgetting system responses. In other words, do not design from Use Cases (Korson, 1998). All Use Cases should be event-response oriented. The objective of any Use Case is to capture what happens when a user does this and the system does that (Rosenberg & Scott, 2001). For example, in the electronic commerce quotation system example, the company had problems creating quality requirements. One of the mistakes they made was focusing too much on required GUIs (Anton, Dempster, & Siege, 2000). Their Use Cases did not detailed what the system
would do, but rather focused too much on design and implementation. Thus, the company had to go back and rework their Use Cases.

Lastly, another common pitfall is writing Use Cases too briefly. All user actions and system responses must be detailed. In addition, do not forget to detail the alternative courses of actions. Sometimes important alternative courses of action are not found until the coding or debugging stage for which the programmer tends to treat them in the most convenient way for him/her. This is necessarily not how the user (i.e. actor) would like to see the system response to them (Rosenberg & Scott, 2001).

**Effective Application of Use Cases**

The application of Use Cases is an efficient and effective technique for collecting software requirements from customers. They help to focus on the real needs and create a shared-vision to all those involved (i.e. customers and software developers). Use Cases are the key to creating quality software (Wiegers, 1997).

Besides avoiding the above pitfalls, Use Cases are effective in assigning priorities to requirements. In fact, to have quality requirements, all Use Cases should be organized hierarchically. There are many reasons for this. First, prioritization helps manage the complexity of requirements. Humans have a limited ability to deal with complexity. Therefore, requirements that are hierarchically structured help the team members to understand them, debug them, and use them to validate the product (Wiegers, 1997).

A second reason for prioritization of Use Cases is it provides a complete set of requirements at the appropriate level of detail. For example, they can be grouped into each domain such as application or architectural. Of course, this implies that each level of Use Case is completed at each domain (Wiegers, 1997).

Another benefit from effective application of Use Cases is it helps to avoid superfluous requirements. When focused on “What do you want the system to do,” every requirement directly supports a specific task the user needs to accomplish with the software system. All too often, customers are tempted to add features that seem like good ideas but are not required for the system to function properly (Wiegers, 1997). In conclusion, Use Cases are excellent for prioritizing requirements and helping to avoid unnecessary requirements. Those are just a few of the effective applications of them.

**Summary**

This academic research paper began with the background and definition of Use Cases. A Use Case, also called a scenario, captures a contract between the stakeholders of a software system about its behavior. Next, a review of the various Use Cases modeling approaches was detailed. The review briefly discussed Adaptive Use Cases, Agent Based Use Cases, and Change Cases.
Outlining two real life examples of Use Cases followed the above review. The examples were a chemical tracking system and an electronic commerce quotation system. Next, the common Use Cases pitfalls as well as effectively applying them were briefly described. Some of the pitfalls to avoid were generating too many Use Cases, forgetting to detail system responses, and being to brief in writing them. Two of the most effective application of Use Cases were prioritization of requirements and the avoidance of superfluous requirements.

Use Cases are increasing in popularity. This is because they tell coherent stories about how a software system behaves in an easy-to-understand format. Therefore, users are able to visualize what the software system will do and not do. Since Use Cases are done during the requirements gathering phase, which is very early in the SDLC process, it allows the changes to be made without massive rework (Cockburn, 2000). This is one of the greatest ways Use Cases add value to any SLDC process. Thus, Use Cases are a powerful user-centric tool that is applicable for any software development methodology. They reinforce the ultimate goal of any software product - to create quality products that let customers to useful work (Wiegers, 1997).
References


Appendix A

Use Case Examples
Figure 1. Sample Use Case Diagram (Fowler & Scott, 1997)
USE CASE 1: BUY STOCKS OVER THE WEB
Primary Actor: Purchaser
Scope: Personal Advisors / Finance package ("PAF")
Level: User goal
Stakeholders and Interests:
Purchaser - wants to buy stocks, get them added to the PAF portfolio automatically.
Stock agency - wants full purchase information.
Precondition: User already has PAF open.
Minimal guarantee: sufficient logging information that PAF can detect that something went wrong and can ask the user to provide details.
Success guarantee: remote web site has acknowledged the purchase, the logs and the user's portfolio are updated.
Main success scenario:
1. User selects to buy stocks over the web.
2. PAF gets name of web site to use (E*Trade, Schwabb, etc.) from user.
3. PAF opens web connection to the site, retaining control.
4. User browses and buys stock from the web site.
5. PAF intercepts responses from the web site, and updates the user's portfolio.
6. PAF shows the user the new portfolio standing.
Extensions:
2a. User wants a web site PAF does not support:
2a1. System gets new suggestion from user, with option to cancel use case.
3a. Web failure of any sort during setup:
3a1. System reports failure to user with advice, backs up to previous step.
3a2. User either backs out of this use case, or tries again.
4a. Computer crashes or gets switched off during purchase transaction:
4a1. (what do we do here?)
4b. Web site does not acknowledge purchase, but puts it on delay:
4b1. PAF logs the delay, sets a timer to ask the user about the outcome.
4b2. (see use case Update questioned purchase)
5a. Web site does not return the needed information from the purchase:
5a1. PAF logs the lack of information, has the user Update questioned purchase.

Figure 2. Sample Use Case Text Form (Cockburn, 2000)