Patterns for Essential Use Cases

Robert Biddle, James Noble, Ewan Tempero
Computer Science,
Victoria University of Wellington, New Zealand.
{robert,kjx,ewan}@mcs.vuw.ac.nz

July 24, 2001

Abstract

Essential use cases are an effective way to analyse the usability requirements for a system under development. Essential use cases are quite stylised — writing good essential use cases is somewhat of a secret art. This paper casts the basics of writing essential use cases into the pattern form. Readers of this paper will be able to write better essential use cases quickly, making it easier to specify usable systems.

Introduction

Systems need to be usable. If people can't use systems we design, they will avoid, circumvent, disparage, and sabotage them.

In the good old days of computing, people were so pathetically thankful to have any kind of computer system at all that they were quite happy to wait in long queues, pick up printouts several days after their jobs were submitted, type programs on chiclet keyboards, and do all sorts of stupid stuff. Unfortunately for us development types, these days are over. In an increasingly large number of systems, the usability of a system is paramount: *If you build it, they won't come if they can't use it.*

This lesson has been writ large recently following the failures of several high-profile internet commerce web sites — if the site isn't usable, no-one will use it. But it holds true even for administrative systems established by government departments or large corporations or consumer and embedded systems: if using the system requires a lot of effort, the people who need to use it will find some other way of achieving their goals, often at your expense.

The discipline of Usage-Centred Design has been introduced to incorporate usability into software engineering development processes. Described in Constantine & Lockwood's *Software for Use* [8], Usage Centred Design is based on *essential use cases*, and draws from ideas from object-oriented methodology [11, 15, 7, 2] as well as task analysis and prototyping techniques common to human-computer interaction designers. A key feature of Usage-Centred Design is that the design practitioner acts as an advocate for users, ensuring concern for usability is maintained throughout the development cycle.

Essential use cases are quite stylised, and writing good essential use cases is somewhat of a secret art. This paper casts the basics of writing essential use cases into the pattern form. The patterns are divided into two groups. The first group is presented in full detail and consists of six basic patterns. The first four patterns describe how to identify the actors in a system, and find and prioritise use cases. The next two patterns describe how to write dialogues for each use case, and how to check those dialogues using role-plays. Figure 1 summarises the problems dealt with by this collection of patterns, and the solutions they provide. In the interests of space, only the bare bones of the second group of patterns are presented. These cover the mechanics of writing essential use cases, organising them, and finding them.

The content of these patterns is not novel, rather, this paper is an attempt to cast some of the techniques of Usage-Centred Design (drawn particularly from *Software for Use*) into a pattern form.

Example

The patterns in this paper use examples drawn from a simple booking system for an arts centre. The initial brief for this system is as follows:

> *Design a program for a booking office of an arts centre. There are several theatres, and people may reserve seats at any theatre for any future event. People need to be able to discuss seat availability, where seats are located, and how much they cost. When people make a choice, the program should print the*

Pattern	Problem	Solution
Actors	Where to you start use-case modelling?	Start with the people (and other systems) who will actually use the system.
Candidate Use Case List	How do you determine what the system should do?	List **Candidate Use Cases** for each Actor.
Focal Use Cases	How can you manage a large number of candidate use cases?	Choose **focal** use cases to drive the design.
Use Case Diagram	How do you know when your candidate use case list is complete?	Draw a use case diagram to show how actors and use cases are related.
CRUD Use Cases	How do you get a complete set of use cases?	Apply CRUD analysis to each of the appropriate domain concepts.
Reporting Use Cases	How do you get a complete set of use cases for reporting?	Ensure you have at least twenty reporting use cases.
Use Case Dialogues	How can you describe what each use case involves.	Write essential use case dialogues for each use case.
Use Case Roleplay	How can you check that use case dialogues are correct?	Act each use case before an audience of the development team.

Figure 1: Summary of the Patterns

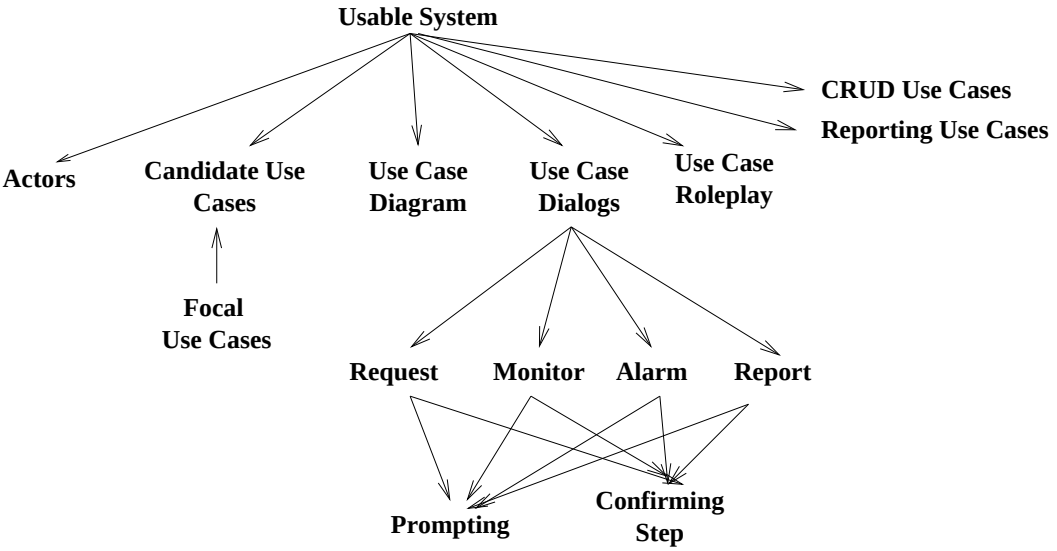

Figure 2: The beginnings of a pattern language. The arrows show the *uses* relationship between patterns. **Usable System** represents the whole set of patterns.

> *price, record the selection, and print out a ticket.*

Typically, we would expect to have much more information (either more text, or at least the opportunity to talk to the project sponsor). We will introduce more details as the example progresses.

Form

The patterns are written in modified electric Portland form. Each begins with a question (in italics) describing a problem, followed by a bullet list of forces and discussion of the problems the pattern addresses. A boldface "**Therefore:**" introduces the solution (also

italicised) followed by the consequences of using the pattern (the positive benefits first, then the negative liabilities, separated by a boldface **However:**), an example of its use, and some related patterns.

Known Uses

It is standard to list known uses for each of the patterns. In the case of our patterns, the known uses are all much the same so we have elected to discuss them here.

The patterns we describe have shown up in a number of projects we have been involved in, including Siemens Step7Lite project (for a programming environment for programmed logic controllers), projects for managing telecommunications plant and managment of service requests, and industry and academic courses.

1 Finding Use Cases

The first patterns describe how to find the use cases in your system. The first four patterns are about exploring the territory, making a rough map of the ground which you will cover in more detail using later patterns. These patterns are also about scoping — making decisions about what is (and what is not) inside the system to be built. The last two patterns are really only proto-patterns. We have noticed it is quite easy to miss use cases for the really common, almost boring, situations, and we have found these two patterns to be useful in avoiding this kind of mistake.

1.1 Actors

Where do you start use-case modelling?

- You have to start modelling somewhere.
- There may be many stakeholders involved in the development.
- Time to market may require very rapid development.
- Technological issues may be overriding.
- It may be very important that you interface to legacy systems.
- Stakeholders may have different priorities than users.
- Use cases are based on user's needs.

Every analysis, design, or modelling exercise has to begin somewhere, however, it is often not obvious where you should start. *Begin at the beginning* is very fine advice if you are telling a story or reading a novel; but the "beginning" of a development project is not necessarily the best place to start modelling for that project. Typically, projects begin when one or more *stakeholders* agree upon the need for development, but the needs or dreams of the stakeholders may not be a good place to start. For example, they may specify particular, detailed solutions (*"the booking system should run on those new WAP phone computers from Nokia"*) rather than the real requirements (*"theatre session times must be accessible over the internet"*).

More seriously, although for political reasons the stakeholders may nominally agree on the importance of a particular project, they may strongly disagree on what the system should be for, what it will do, what is required for it to be a success, and so on. Stakeholders can also booby-trap a development project before it gets started by insisting that time-to-market requirements preclude any analysis, modelling, or design; or the system may need to interface to fickle external systems; or meet strict technological or resource challenges.

Therefore: *Start with the people (and other systems) who will actually use the system.*

One of the most important tasks in defining a system is to work out what the system is (and conversely, what it is not). We do this by considering the **Actors** of the system, that is the people (and other systems) that are *outside* the system we will build, but that interact *directly* with it.

First, by brainstorming, textual analysis, interviewing clients, and similar activities, come up with a list of the kinds of people who will use the system. Once you have the list, briefly determine the characteristics of each user. Most especially, you need to attempt to understand users' goals or intentions when using the system, and then characterise different kinds of actors in detail. For example, consider:

- actors' knowledge of the domain
- their expected knowledge of the system you will design
- whether they will use the system often or seldom
- any special support requirements

If external systems are important, you should also list the **system actors**, that is, the important systems to which you have to interface. You should then characterise the system actors, describing their characteristics, typically by referring to existing manuals or protocol definitions.

Finally, you should roughly prioritise the actors in terms of their importance to the system as whole.

Example

In considering who would be using the Arts Centre Booking System (ACBS), an important issue immediately becomes apparent: will this system only be used by Arts Centre staff, or will it be available to customers (for example, as a kiosk or web site)? The answer to this question will significantly impact the nature of the system, and so it is best that it be answered quickly.

In the case of our example, there is not enough information to answer this question, and so we would have to go back to the sponsor to find out. We will assume that the system will only be used by Arts Centre staff.

Just listing the staff in the Arts Centre gives a good idea of the likely actors. Such a list might include: the people at the ticket booth who actually sell the tickets (ticket sellers), the person who cares about how well events go, such as the attendance rates for performances (business manager), the person who is in charge of what events are on and what performances there are (event manager), the person in charge of the arts center (managing director), the person in charge of finances (accountant), and possibly other people in the organisation (administration staff). And of course there are also the arts centre patrons, who clearly have an interest in what the system does. From this list, we can come up with a first cut at our list of actors.

Ticket Seller: This is the person who sells tickets, makes reservations, and answers customers' queries about events in the Arts Centre. People playing this role will often be casual staff, and so cannot be assumed to have much knowledge of the domain. On the other hand, they are employees and so some minimal level of domain knowledge can be assumed (for example, through staff training). They will not be expected to know anything about computer systems, but they will use the system very frequently.

We have already determined that arts centre patrons will not directly use the system, however they will have expectations of the ticket seller, which will translate to the ticket seller's expectations of the system.

Event Manager: This is the person who decides what events are booked into the arts centre, when performances happen (or not), where they are held, and what the seating layout is. People playing this role will have a lot of knowledge about events and related aspects of the job, but cannot be assumed to have much knowledge about computer systems. They will use the system several times a week, but probably not as often as once a day.

Business Manager: The business manager probably doesn't want to touch the system at all, and so will get someone else to actually produce the reports he needs from the system. Nevertheless, this does not mean there is no Business Manager Actor, but rather there is someone playing that role. For that reason, either this actor will not have much knowledge about the domain (in this case, what the business manager's concerns are), or will not have much knowledge about the system (or possibly both).

Accounting System: By talking to the Accountant, we discover that her only interest in the proposed system is to get the sales information, and by preference would like it to be delivered directly to the existing accounting system. Assuming this system has the ability to interface to other computing systems, it would be a **system actor** for the proposed system.

Looking over the list, we can see that it is already rougly sorted in priority order.

Consequences

Just determining who are the actual users of the system can answer some important questions about what the system will have to (or not have to) do. Finding out who else cares about what the system does, even if they will never use it directly, can also quickly identify important functionality. Very few systems are built to be independent of any existing systems, and interfacing with existing systems is often the source of many frustrating problems, so it is important to identify these systems early. What the client thinks is important is, of course, important to you, but if this is not what the users want, then you should know about it as soon as possible.

However:

Identifying actors takes time and effort, not only from modellers but also from stakeholders, and, of course, the actual users. Getting access to users can be difficult, especially if they are not employees of the stakeholders' institutions, and many actors will not relish being the objects of analysis. Modelling users can also irritate stakeholders if they don't consider users a high priority. They may much prefer their money was spent on something useful, like programming.

Discussion

Note that we consider only the so-called *direct actors*, that is actors that use the system itself. Often there can be *indirect actors* who use the system on behalf of another person; we may note these down but don't concern ourselves primarily with them. Conversely, a shallow analysis can miss *hidden actors*, such as users that install and maintain the system, who may not be part of the main purpose of the system but who will interact with it directly.

Starting with actors may help result in a usable system, but may do little to placate stakeholders. Some kind of modelling or analysis of stakeholders goals and requirements, and on-going project-wide risk analysis are important practices for all projects, however, they are separate activities from (and do not substitute for) beginning the modelling by analysing the goals and characteristics of the poor people who will actually have to put up with the system as part of their lives.

1.2 Candidate Use Case List

How do you determine what the system should do?

- You have to start modelling somewhere.
- There may be many stakeholders involved in the development.
- Time to market may require very rapid development.
- Users don't describe much about the requirements of a system.
- Stakeholders' ideas of what the system should do are often informally stated.
- The system must meet users' needs.
- Users often cannot say what they want the system to do.
- There needs to be some way to get an estimate of the size of the system for planning.
- There has to be some way to decide what's necessary for the system and just those features that would be nice.
- Stakeholders may have different priorities than users.
- Stakeholders may have unrealistic expectations.
- Technological issues may be overriding.

Users are unfortunately part of the problem, not part of the solution. Knowing that you have to build a system that will be usable by particular users often makes your job *harder* rather than easier, since having to worry about the people that will use a system is just another problem on top of all the technical or managerial issues of making any sort of system work.

So, just knowing the actors doesn't help work out what a system should actually *do*. What you need to know is what the actors need to accomplish with the system, what are their intentions or goals, and what responsibilities are incumbent on the system to support them.

Similarly, lists of requirements ("wish lists") produced by stakeholders or the marketing department are often very informal, imprecise, and irregular, mixing large and small, detailed and vague, important and irrelevant information all in one document.

Of course, stakeholders still want you to stop mucking about and deliver the system yesterday.

Therefore: *List **Candidate Use Cases** for each Actor.*

Consider each Actor in turn, starting with the highest priority actor, and write a list of posisble (candidate) use cases for each actor.

A use case is one complete case of system use. In other words, a use case describes a single sequence of interaction between an actor and a system. From the actor's point of view, the new system should seem to be a "black box", which exhibits only behaviour, with no internal workings visible. A candidate case should be short and sweet, with just enough to be meaningful. Some examples are:

- making text bold
- printing out your work
- rebooting a PC

Any non-trivial system will have a lot of use cases, and it will take a while to nail down which ones actually apply to your system, so identify as many *candidate* use cases as quickly as possible. A candidate use case is just what it says — something that may become a use case, but there's no commitment to it being so at this stage.

Actually finding or determining the use cases is not easy, and involves all the vagueness and uncertainty of analysis. To find use cases, you can use domain knowledge, textual analysis of wish lists or other documents, standards, other systems, and interview stakeholders and people working in the domain. Most importantly, you must look at the potential users of the system (already modelled as actors) and try to understand them and their goals and intentions.

Example

Looking at the text and our list of actors, we can quickly come up with an initial set of use cases:

Ticket Seller: Issue ticket, Show seat availability, show seat location, show seat price

Event Manager: Add event, Schedule performance, Modify performance information

Business Manager: Print report

Accounting System: Produce sales information

This is just a first cut, based on the text we had available. We can easily add to this list, for example by applying the patterns in section 3. In doing so, we would also consider renaming what we have to make them more consistent. A larger set of use cases is shown in figure 3.

Consequences

A list of use cases can give you a fairly good idea about what the system needs to do, and it usually fairly easy to quickly come up with a set that's representative of what's actually wanted by the stakeholders. Finding use cases should involve stakeholder and user team members, and incorporating these people into the process has the advantage that they will become better disposed towards the project (provided they are treated with a modicum of respect).

Use cases are informal enough to allow good communication with the stakeholders, giving them the feeling that they understand what the system will actually do. This increases their confidence in the project. Use cases are also quite specific in detailing what the system has to do, thus reducing misunderstanding and ambiguity that is often associated with informal requirements.

The list of use cases also gives a good idea of size of the system overall. Because each use case should have the same granularity, describing roughly the same "amount" of interaction with the system, a collection of use cases will give a better idea of the complexity of a system than a list of randomly-sized requirements. They can also be used to derive test cases, to estimate effort (by tracking the number of use cases completed versus time spent in any stage), and to guide documentation.

However: identifying candidate use cases for actors does take time and effort away from more obviously "productive" development or from users' and stakeholders' revenue-earning work. Listing use cases can seem pointless to stakeholders who already "know what the system should do!" especially if they already have other kinds of lists of requirements for the system, and if it turns out that those lists are wrong.

See Also

Use cases were first described by Jacobson for describing Danish telecommunications systems at Erricson [11]. There are number of other books describing use cases and their use in software development [13, 7, 1]. Other related patterns are listed in section 3.

1.3 Focal Use Cases

How can you manage a large number of candidate use cases?

- Even a small system can have a large number of candidate use cases.
- Some use cases will be central to the system while others are only peripheral
- Different use cases can take more (or less) effort to implement.
- Different use cases can be more (or less) risky to implement.
- Some use cases are more important to users than others
- Some use cases are more important to stakeholders than others
- Some actors are more important to to stakeholders others

Candidate essential use cases are quite small, each describing one course of use of a system. Because of this, there can be a large number of them, perhaps 40-50 for a small system, and 200-300 for a medium sized system. This raises another problem: how do you manage and prioritise these use cases. In particular, how do you know where to start with the next part of design. Some use cases are more equal than other use cases. They may take more time (or impose more risk) to implement, they may be more important to actors (say because they will be performed more frequently than other use cases), or they may be more important to stakeholders (for their own impenetrable reasons). How can you placate the developers (who already think this is too big and too hard to build) while still honouring the stakeholders (who are paying for this, after all).

Therefore: *Choose **focal** use cases to drive the design.*

It's not easy choosing what to work on first. Every-

one has their own idea of what's important. That's why we don't say "important", we say "focal": we choose to focus on these use cases to drive the design. Focal use cases are typically those that are the most important to users, but also include use cases to cover the main responsibilities to the stakeholders and cover risks expressed by development.

To identity focal use cases, you can print out the list of every use case, and then rapidly work through the list several times, each time giving each use case a score (say from 1 to 5) for one particular aspect of importance, such as: frequency of use, importance to stakeholders, risk to development, actor priority etc. Several developers can quickly rank each aspect in parallel, although its useful to have two or three estimates of each aspect. Then, add up the scores for each aspect, sort the use cases on different combinations of aspect scores (a spreadsheet is helpful here) and then choose the order that seems to make the most sense. The top 10% of use cases (to a maximum of 20) are your focal use cases.

In making this decision, it can be useful to work towards a minimally useful system, that is, one that can be useful to some of the users. The reason for this is that some use cases identified as focal may depend on other use cases. For example, "purchase ticket" cannot be done without any tickets to purchase, implying that a use case like "add event" will be needed as well. On the other hand, if you are planning an iterative development, it may not matter if intermediate iterations of the system are not minimally complete, as long as the functionality can be supplied in later iterations.

Example

Even a simple version of our Arts Centre Booking System could have 50 use cases, but a version that might still be useful may only need to implement a dozen of them. For example:

Ticket Seller: List Event Performances, Purchase Tickets, Report Availability of Seats, Report Event Details, Show Location of Seats

Event Manager: Add Event, Schedule New Performance

This doesn't allow reserving of seats, cancelling of events, or reporting on ticket sales, but would still give a very good idea of what the final system might be capable of.

Consequences

Ranking use cases and finding focal use cases gives you a good idea about where to go next in your design. It helps in reducing the risk by concentrating on use cases that are most likely to produce a design that will be of most use. The non focal use cases will either not impact the design much when implemented, or will not impact the usefulness of the system if not implemented.

However: Prioritising use cases can give a false sense of security: the system is described by all the use cases, not just the focal ones. Making some use cases a higher priority implies making others lower priority, and risking alienating any stakeholders who champion those use cases.

See Also

The Extreme Programming Planning Game [4] is an incremental take on the same idea, in which programmers and customer representatives (stakeholders) constructively argue over which use cases to prioritise in any given iteration. From a usage-centered perspective, there is a danger in this process: if the customer is not a user, then noone represents the interests of the users — in the same way that noone represents the interests of a child while its mother and father argue about the divorce. An important, explicit responsibility of a usage-centered design practitioner is to balance the competing interests of developers, stakeholders, and users — with the key priority going to users.

1.4 Use Case Diagrams

How do you know when your candidate use cases list is complete?

- You can keep modelling forever, but clearly with diminishing returns.
- If you stop too soon, you may miss important things.
- If you stop too late, you waste resources.
- It's easy to get lost in the detail of the models you are building.
- You need to convince stakeholders and other team members that the modelling is done.

Systems analysis would be a great job if we never had to deliver anything. On a project of any size, you can keep modelling forever, but (seen from outside) continued modelling has clearly diminishing returns.

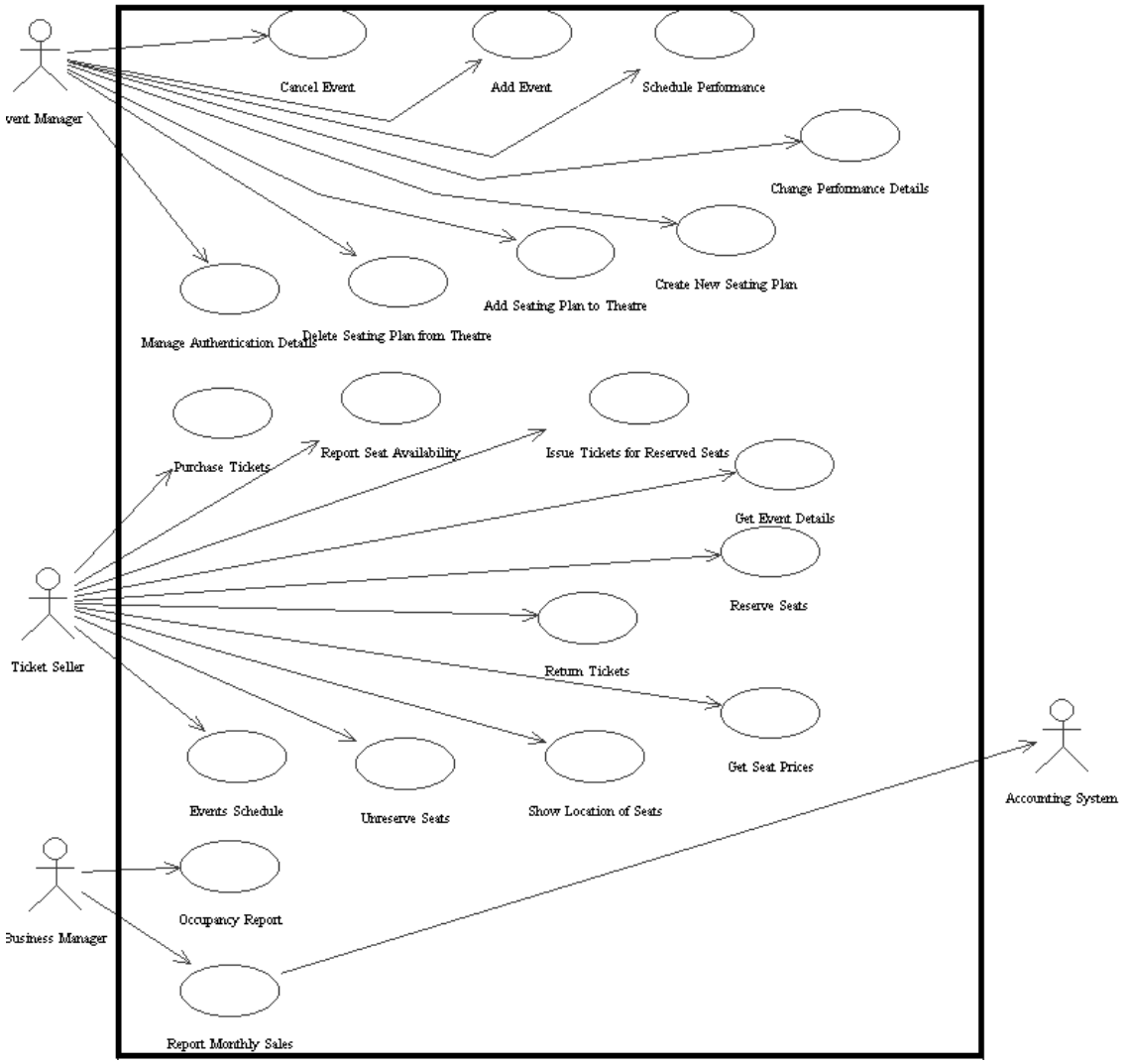

Figure 3: A use case diagram, showing the system boundary.

Stakeholders don't want to pay for unnecessary modelling — but then again, they may consider *any* modelling unnecessary. So, how can you know when you have enough the use cases — so that you can stop? Conversely, how do you know when you don't have enough use cases — so that you don't end up revisiting obvious things that you've missed? Answers to these questions will need to convince other teams members, stakeholders, and so on, so how should you present you use case models to make these arguments?

Therefore: *Draw a use case diagram to show how actors and use cases are related.*

A use case diagram shows stick figures for the actors in with the system, and ovals for each use case. We write the descriptions of the users beside them, and the names of the use cases inside the ovals. The users involved in particular use cases are connected to those cases by lines. Draw a box (the "system box") outline around all the use cases, to make clear the boundaries of the system to be designed.

A use case diagram is quite simple, but can serve a subtle purpose. By depicting the users and the set of use cases, the diagram can be a useful focus for activities to check the use case model. The box around the use cases makes the "black box" nature of the system clear, and the lines between users (outside the system) and use cases (within the system) highlight the ways in which users interact with the system.

Using the diagram, explicitly ask yourself the following questions:

- Is there an actor representing every kind of user who will use the system?
- Is there a system actor for every external system with which this system needs to communicate?
- Can each actor do everything they need to do using only the use cases they are related to?
- Are any obvious use cases missing? For example, use case models are often symmetric: if there are use cases for creating bookings, printing booking receipts, printing performance receipts, and cancelling performances, perhaps there should also be use cases for cancelling bookings and creating performances.

Unless you are on a small system (if you have not more than 15-20 use cases) draw one use case diagram for each actor (or for a few related actors), rather than one diagram for the whole system.

Example

Figure 3 shows a use case diagram for the Arts Centre Booking System.

Consequences

A use case diagram provides a gestalt view of the system, showing not just the parts of the system, but also gives a feel for how the parts might interact. It is also useful for new team members coming onto a project, and convincing stakeholders who have problems with written documents but like pictures. More importantly, the process of drawing and staring at a diagram can help you get to grips with the model in its entirety, to find missing or duplicate use cases, missing actors, and so on. Large organisations with formal development processes or ISO certification typically require sign-offs and these diagrams can prove convincing here.

However: Models are never really complete, so drawing diagrams may again give a false sense of security. Drawing pretty diagrams can become an end in themselves, rather than a tool for assisting modelling, especially if you are proud of your prowess with a CASE tool.

See Also

UML Distilled [9] briefly introduces use case diagrams. Software for Use describes more complex use case diagrams in more detail [8]. Jacobson et al. use a system box [11].

1.5 CRUD Use Cases

How do you get a complete set of use cases?

- There are many use cases, even in a small system.
- Many use cases are infrequently used and "non critical" to understanding what the system will do in general, but are nonetheless necessary for a complete description.

Even for a small system, there can be a larger number of use cases than you might initially expect. A common mistake for people to make is to identify one use case they know they need, but miss related use cases. For example, there may be a use case to display all the details about a particular kind of record, but no use cases that actually create records. They will also completely miss a set of use cases that apply to a concept in the domain model.

Therefore: *Apply CRUD analysis to each of the appropriate domain concepts.*

Look at the use cases you have, and determine whether any of them correspond to a **Create**, **Read**, **Update**, or **Delete** (CRUD) of class in the domain model. If you find any in this category, check whether any of the other CRUD use cases are needed.

Now consider the rest of the classes in the domain model and check with they should also have CRUD use cases.

Rename these uses cases if the standard names don't make sense.

Example

Consider "Event" in the ACBS. One reasonable use case would be "Display Event Details". Structurally, this is "**Read** Event". If we have this as a use case, then we might want the following: "**Create** Event", "**Delete** Event", and "**Update** Event".

There may also be related use cases. For example, another way to **Create** something is to **Copy** it (e.g., "Reschedule Event"), and there may be more than one way to **Delete** something (e.g., "Cancel Event", where the Event details are not actually removed from the system).

It might make sense to rename some of the use cases ("Modify Event details" instead of "Update Event").

Consequences

Applying CRUD analysis can quickly get many relevant use cases without requiring a lot of effort. In fact, CRUD analysis is amenable to automation.

However:

You can quickly get a huge number of use cases, many of which are not immediately needed (and so should receive low priority). For example, Delete use cases can often be left out of early releases.

Not all of the CRUD use cases are needed for every concept so CRUD analysis should not be applied blindly.

Not all classes in the domain model should have CRUD analysis applied to them. For example, "Time Period" may be a sensible class to have, but this concept is never required on its own in the application, so there is no need for use cases just to deal with it.

See Also

Alistair Cockburn talks about CRUD use cases and how to present them. He advocates grouping them all as a single use case, e.g., a "Modify" use case [7]. We prefer to analyse each use case individually because even trivial CRUD use cases should represent a valid use of the system, and because each use case can be measured and tracked individually throughout development.

1.6 Reporting Use Cases

How to you get a complete set of use cases for reporting information?

- Reporting is very important for lots of systems
- Reporting is boring for implementors, so they can underestimate the effort required to produce reports.
- Reporting can be boring to model, too.
- Reporting is often boring to many of the users or stakeholders, while simultaneously crucial to the others.
- Reporting often has value for indirect or hidden actors

Therefore: *Ensure you have at least twenty reporting use cases.*

Even though reporting is boring, if it's important to someone, you have to model it. We suggest analysing the problem, domain model, brainstorming, and so on, until you have at least twenty reporting use cases. This number is arbitrary, but is sufficiently large that you should get a feeling for most types of reports the system must produce. Thus, you are unlikely to miss important type you capture all the important cases someone will need to report on, and ensure your effort estimates are correct if you are using use cases to drive estimation. Often, after struggling to find twenty reports, the next thirty come very easily!

Example

Figure 3 only has two reporting use cases ("Occupancy Report" and "Report Monthly Sales") although some others may also be regarded as reports (for example, "Report Seat Availability". Other reporting use cases might include: "Occupancy Report by Theatre", "Occupancy Report by Event, "Occupancy Report by Performance", "Occupancy Report by Week", all of which would be of interest to the Business Manager. However similar kinds of reports (almost certainly organised differently) would be useful to the busy Ticket Seller having to answer questions of the form "Are the plenty of free seats for tomorrow's performance of Dracula?".

Consequences

Actually finding 20 reporting use cases isn't necessary, the main point is that just being force to try to come up with 20 (or whatever the number really is) reporting use cases almost always produces use cases that weren't already listed, but instantly recognisable as crucial to the system.

However:

Sometimes it is too easy to come up with lots of reporting use cases, when only some of them will actually be needed in the system.

2 Detailing Use Cases

Once you have a list of candidate use cases, then you need to go through each one and describe them in detail. Of course you start with the focal ones first — indeed, you can detail the focal cases before you've finished the "whole" use case model.

2.1 Essential Use Case Dialogues

How can you describe what each use case involves.

- You have a prioritised list of actors — but this doesn't help you work out what the system should do.
- You have a list of candidate use cases — but this list doesn't provide much detail. How can you tell what needs to be implemented for each use case.
- You don't have the details of the system's design because you haven't designed it yet.

- You don't want to spend too much time or effort writing useless documentation.
- You don't want to limit your implementation options.
- Time to market may require very rapid development.
- Technological issues may be overriding.
- Abstraction is a difficult, learned skill.

Even when you've completed your **Candidate Use Case List (1.2)**, identified the **Focal Use Cases (1.3)**, and perhaps checked some **Use Case Diagrams (1.4)**, you still don't really have much detail about what the use cases will involve: what information needs to be provided by actors, and what behaviour needs to be provided the system to successfully implement the use case.

The names of use cases only give you a rough idea of what the system is supposed to do. You still need to determine the detail of the interaction with the system. However you don't want to have to make decisions relating to technology (such has what I/O devices will be available, user interface requirements, and so on), despite pressure by the stakeholders to use particular technology (they will probably change their minds by the time implementation starts). And on top of that, you still have to come up with the details quickly.

To address these issues, you need to provide more detail about each use case; but, how much detail is too much? You could write detailed descriptions of what each use case will involve, but this will take lots of effort, produce a large amount of dense documentation that will be hard to manage, and probably of little use to the eventual development team.

Furthermore, to be able to write detailed descriptions of the interaction of each use case requires that you have already decided how each use case will be designed, if not already implemented — whether this will be handled by a computer system, by a worker as part of a business process, by an application program, a web site, or a WAP phone. Unfortunately, if you are solely responsible for analysis (say the design is being outsourced to a graphics house and the implementation to India), then you don't want to have to do design that will be replaced later. If you are responsible for design, you can't really begin that until you have worked out what goals the design should meet to be usable — that is, what users need to do to complete each use case.

Therefore: *Write essential use case dialogues for each use case.*

Essential use cases are part of Usage-Centered Design, as developed by Larry Constantine and Lucy Lockwood [8]. The term "essential" refers to essential models that "are intended to capture the essence of problems through technology-free, idealized, and abstract descriptions". Constantine and Lockwood define an essential use case as follows:

> *An essential use case is a structured narrative, expressed in the language of the application domain and of users, comprising a simplified, generalized, abstract, technology-free and implementation independent description of one task or interaction that is complete, meaningful, and well-defined from the point of view of users in some role or roles in relation to a system and that embodies the purpose or intentions underlying the interaction.*

Constantine and Lockwood give the examples shown in figures 4 and 5. The dialogue in figure 4 is for a conventional use case, described in terms of actions and responses. The dialogue in figure 5 is for an essential use case, described in terms of intentions and responsibilities. The steps of the essential use case are more abstract, and permit a variety of concrete implementations. It is still easy to follow the dialogue, however, and the essential use case is shorter.

So we document each use case with a "use case dialogue". We write use cases on index cards, so we also call these dialogues "use case cards". A use case dialogue documents the chronological steps in the use case as the user and the system interact. We typically document the use case card with the users part on the left hand side, and identify this as the "user intention", which reminds us to focus on the users real goals for the step. On the right hand side, we identify the system "responsibility", stressing that the system too has goals incumbent upon it. The division down the centre can be regarded as the "interface" between the user and the system, and serve as reminder that interaction is communication across this division.

We write the steps of the interactions under the assumption that the actor has already chosen to do this use case and has already told the system that the are doing it, so we don't need to include a separate step to start the use case.

We prefer essential use cases to conventional use cases because they allow a certain independence from technology choices in later or subsequent implementation, and also allows us to make progress quickly, without having to make difficult decisions otherwise necessary. Also, we believe their emphasis on system

gettingCash

User Action	*System Response*
insert card	
	read magnetic stripe
	request PIN
enter PIN	
	verify PIN
	display transaction menu
press key	
	display account menu
press key	
	prompt for amount
enter amount	
	display amount
press key	
	return card
take card	
	dispense cash
take cash	

Figure 4: A conventional use case for getting cash from an automatic teller system. (From Constantine and Lockwood.)

gettingCash

User Intention	*System Responsibility*
identify self	
	verify identity
	offer choices
choose	
	dispense cash
take cash	

Figure 5: An essential use case for getting cash from an automatic teller system. (From Constantine and Lockwood.)

responsibility leads to better traceability between requirements and design.

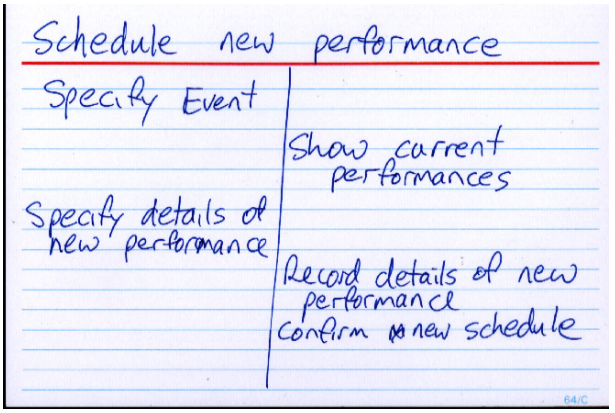

Figure 6: A use case card showing the essential use case details for Schedule new performance.

Example

Figure 6 shows an example of a use case card.

Consequences

Essential use cases dialogues capture the core requirements of each use case, but without getting into technological details. Because of this, they are short, quick to write, and easy to manage.

Essential use cases are smaller (and thus quicker to write, review, and modify) than longer, more detailed use cases (for example, the more traditional use cases used in the Rational Unified Process [10]).

However:

You still have to write them, which takes time and effort. Finding the "correct" level of abstraction in which to write a use case — enough detail so that it makes sense, but not too much so that it determines the details of the interface design — can be difficult, and so can take several attempts for some use cases.

See Also

Writing dialogues can lead you to revise the list of use cases and use case diagrams. Consider the bodies of each use case — if two use case bodies are the the same, they should probably be the same use case, so eliminate one of them. If one case seems to need more than one body, you probably need different use cases. Two use cases that are similar can be modelled by **Specialisation (4.3)** or **Inclusion (4.2)**; the possible errors that can occur during a use case can be modelled by **Extensions (4.1)** or **Conditions (4.4)**.

Use case cards were inspired by CRC cards [3, 14, 5]. They are also similar to the Story Cards used to

schedule Extreme Programming iterations [4]. Wirfs-Brock introduced the idea of the two-column format [16].

2.2 Use Case Roleplay

How can you check that use case dialogues are correct?

- Use cases dialogues need to be correct and consistent.
- Incorrect use cases can waste development effort.
- Every team members needs a shared understanding of the use cases.

Once you have written some essential use cases, you need to verify that they make sense, that they describe all the communication that is needed for an actor user and the system to carry out the use case, and that they don't include any unnecessary implementation details. You don't just write use cases for the fun of it: the point of the use case model is to direct the development effort, so inconsistencies or errors in use cases can cause problems if they are not caught later on. It's important different team members have the same understanding of use case dialogs, or inconsistencies and errors are more likely to be introduced.

Therefore: *Act each use case before an audience of the development team.*

In a use case roleplay, one person takes the role of the user, and another person takes the role of the system. They then proceed to act out the interaction, using the use case body as a script. Other people critically observe the role play. Although use cases should not be very long, use case roleplay is quite useful for checking the use case.

There are several things to watch for in use case role play. One is continuity, to make sure that both user and system understand when they have something to do, and to make sure they understand what needs to be done. Confused pauses can indicate misunderstanding, which can often highlight unresolved issues in describing the use case. Another thing to watch for is assumed information. Sometimes the user or the system will mention information they are relying on, yet would not actually know. It is important to check these details, because they can again show that the use case has not yet been fully described.

Example

The following gives a representative example of how a roleplay proceeds. In particular it gives examples of the kinds of errors that crop up.

Report Seat Availability

The scene: The ticket seller ("user") is using the computer "system" to determine whether the seats requested by the Arts Centre patron for a performance are in fact available.

Take 1:

User: I say which performance I want and the system shows me the performance details.

CUT! — it's the system's job to say what the system does. This is often just an error made by the role-player, but can also indicate confusion as to where the system boundary is.

Take 2:

User: I say which performance I want.

System: I display the performance details and say whether or not the seats are available.

CUT! — the seats haven't been specified yet.

Take 3

User: I say which performance I want.

User pauses waiting for a response, then Looks over to the person playing the system, who is still looking at the use case card, and doesn't realise he's being cued.

System?

System: You're supposed to say what seats you want to know about too. *Points at card.*

User: Oh, right

CUT. The roleplay does not allow anyone to hide — all participants have to engage with what the use case is about.

Take 4:

And so on...

Consequences

Use case roleplays highlight problems in your use case dialogues, so you are able to detect and correct them early. The audience of the roleplay can both see how the dialogue should work, and ask questions to ensure everyone understands the use case.

However:

Roleplaying is another checking practice that is subject to diminishing returns: pedants can make

the whole process much more annoying and time-consuming that it needs to be, small errors in use cases are not that bad as they can be easily detected later, and many people object to the ritual humiliation of standing up and performing in front of the rest of the team. These kind of group activities can also be soured by managerial involvement, either by a culture of "enforced fun", or worse by turning internal consistency checks in to excuses for evaluating and firing staff members.

Discussion

Using roleplay to assist use case checking is not strictly necessary, but it does harness several human skills. It uses the abilities of the people playing the roles to identify with the roles, which can often cause them to focus more intently on the user intention or the system responsibility, and to detect problems. It also uses the ability of a critical audience to follow the dialog, and brings into play skills developed in understanding stories. These skills help people to detect discontinuities or assumptions, and so detect possible problems in the use case. Use case roleplay makes adds more fun and variety into the activity, and these also heighten attention.

See Also

Use case roleplays were inspired by CRC Card Roleplays [14, 5]. Further discussion on use case roleplay can be found in [6]. Our use of roleplay is similar to Wirf-Brock's use of "conversations" to evaluate use cases [17].

3 Use Case Dialog Patterns

Once you start writing use cases you'll realise that lots of kind of use cases come up over and over again — that there are actually *patterns* in the dialogue bodies of essential use case themselves. This section lists a number of these patterns. In the interests of space, we give only the bare bones of each pattern.

3.1 Alarm Use Case

How do you have the system inform the user about something?

- The system needs to draw actor's attention to a change in its internal state.
- The system is about to break a business rule.
- The notification should be asynchronous, that is, actors should not have to trigger the use case.

Therefore: *Write a use case that begins with the system taking the responsibility to warn the user.*

Example

Warn of start of performance

User Intention	System Responsibility
	Signal "performance about to start"
	Show name, theater, and times of performance

Consequences

- The system takes responsibility for initiating the use case.
- The system can pass information about the alarm to the actor.
- The actor does not have to interrupt their current task immediately to respond to the alarm.
- The actor can ignore the alarm.

If the alarm is important, you may need to include a **Confirming Step (3.6)**:

Warn theater performance undersold

User Intention	System Responsibility
	Signal "performance undersold"
	Show name, theater, time or performance, and percentage of seats sold
Confirm warning	

This variant has the following different consequences to the main pattern:

- The actor cannot continue with their current task: they must interrupt it to confirm the alarm.
- The actor cannot ignore the alarm.

Alarm use cases can often indicate (potential) violations of business rules — say that a performance should not continue if less than 15% of seats have been sold by the time it starts.

3.2 Requesting Use Case

How do you write a use case when the user needs to know something from the system?

Therefore: *Write a use case where the actor describes the information they require, and then the system presents that information.*

Example

Get Seat Prices

User Intention	System Responsibility
	Offer performances
Choose performance	
	Show prices for chosen performance

3.3 Monitoring Use Case

How do you write a use case where the user often needs to know about a relatively small amount of important information from the system.

Therefore: *Write a use case where the system presents that information.*

Example

Show Today's Performances

User Intention	System Responsibility
	Show today's performances

3.4 Commanding Use Case

How do you have the user get the system to do something?

Therefore: *Write a use case where the user provides information on the request, and the system has the responsibility for performing the command.*

Example

Print performance schedule

User Intention	System Responsibility
Chose start and end dates	
	Print schedule of performances from start to end date

3.5 Prompting Step

How should you write a use case when the system knows some information that would help the use make a decision?

Therefore: *Give the system the responsibility of offering that information before the user makes the decision.*

Example

Reserve seats for performance

User Intention	System Responsibility
	Offer unreserved seats
Choose seats	

3.6 Confirming Step

How should you write a use case when it is important that correct information is communicated between the actor and the system?

Therefore: *Require the actor or system to confirm the information.*

Example

Pay for reservation

User Intention	System Responsibility
	Present reservation details
	Offer payment methods
Choose payment method	
Supply payment details	
Confirm method and details	
	Accept payment
	Confirm booking

4 Organising Use Cases

In this section, we briefly list a number of patterns which will describe how to model relationships between use cases, based on the UML and Usage-Centered Design relationships.

4.1 Extension

How do you model errors and exceptions in use cases?

Therefore: *Use extending use cases.*

4.2 Inclusion

How do remove commonality between use cases?

Therefore: *Make a new use case containing the common steps, and include it in the use cases that have the common steps.*

4.3 Specialisation

How do you handle more general and more specific use cases that do the same kind of thing?

Therefore: *Use specialisation.*

See Also

Prefer inclusion to specialisation.

4.4 Conditions

How do you model use cases than can only operate under certain circumstances?

Therefore: *Use pre- and post-conditions to control when use-cases are permissible.*

See Also

Prefer extensions to conditions. Pre- and post-conditions should match in a complete model.

Conclusions

In this paper, we have presented a number of patterns for writing essential use cases for a system. Many of these patterns may also be applicable to conventional use cases, although we believe the patterns are more evident in the essential form of the use case. Clearly a number of the patterns we have discussed here need more development, and we are investigating other possible patterns.

Acknowledgements

Thanks to Larry Constantine, Lucy Lockwood, and participants from KoalaPLoP 2001 Workshop C — Saluka Kodiuwakku, Pauline Khoo, and John Hosking — for their comments on this paper.

References

[1] Frank Armour and Granville Miller. *Advanced Use Case Modeling: Software Systems, Volume 1.* Addison-Wesley, 2001.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[3] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 1–6, 1989.

[4] Kent Beck and Martin Fowler. *Planning Extreme Programming.* The XP Series. Addison-Wesley, 2000.

[5] David Bellin and Susan Suchman Simone. *The CRC Card Book.* Addison-Wesley, 1997.

[6] Robert Biddle, James Noble, and Ewan Tempero. Use case cards and use case roleplay, 2001.

[7] Alistair Cockburn. *Writing effective use cases.* Addison-Wesley, 2001.

[8] Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design.* Addison-Wesley, 1999.

[9] Martin Fowler and Kendall Scott. *UML Distilled: A brief guide to the standard object modeling language.* Object Technology Series. Addison-Wesley, second edition, 2000.

[10] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[11] Ivar Jacobson, Mahnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[12] James Noble. Classifying relationships between object-oriented design patterns. In Douglas D. Grant, editor, *Proceedings of the 1998 Australian Software Engineering Conference*, pages 98–109, November 1998.

[13] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: A practical approach.* Addison-Wesley, 1999.

[14] Nancy Wilkinson. *Using CRC Cards - An Informal Approach to OO Development.* Cambridge University Press, 1996.

[15] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object Oriented Software.* Prentice Hall, 1990.

[16] Rebecca J. Wirfs-Brock. Designing scenarios: Making the case for a use case framework. *The Smalltalk Report*, 3(3), 1993.

[17] Rebecca J. Wirfs-Brock. The art of meaningful conversations. *The Smalltalk Report*, 4(5), 1994.