Patterns for Usage Centred Design

ROBERT BIDDLE, Carleton University, Canada
JAMES NOBLE, Victoria University of Wellington
EWAN TEMPERO, University of Auckland

Categories and Subject Descriptors: H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Design*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human Information Processing*

Additional Key Words and Phrases: usage centred design

ACM Reference Format:
Biddle R., Noble J., and Tempero E. Patterns for Usage Centred Design. 2017 HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 32 pages.

INTRODUCTION

The discipline of Usage-Centred Design has been introduced to incorporate usability into software engineering development processes. 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. Described in Constantine & Lockwood's *Software for Use* [1999], Usage Centred Design is based on *essential use cases*, and draws ideas from object-oriented methodology [Jacobson et al. 1992; Wirfs-Brock et al. 1990; Cockburn 2001; Beck 1999] as well as task analysis and prototyping techniques common to human-computer interaction designers. The emphasis on tasks, and the use of the system distinguishes usage-centred design from e.g. other approaches that are user-centred, rather than usage-centred.

Essential use cases are quite stylised, and writing good essential use cases is somewhat of a secret art. 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. This paper resurrects two ancient papers, *Patterns for Essential Use Cases* [2001a] and *Patterns for Essential Use Case Bodies* [2002], and updates them to reflect our subsequent experience, and to provide an easily available reference for these techniques, incorporating ideas from Goal Driven Design [Cooper et al. 2014] and Inclusive Design [Hill et al. 2017].

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 responsibility leads to better traceability between requirements and design.

Table I summarises the problems dealt with by this collection of patterns, and the solutions they provide. Figure 1 shows the structure of the patterns. The first pattern introduces the key problem of constructing a usable system, and the following patterns explain how to identify the users of the system and their patterns of use. Subsequent patterns explain how to model essential use cases in increasing detail: writing essential use case dialogues, reoccurring patterns of use cases, steps within use cases, and relationships between them.

Table I. Summary of the Patterns

Pattern	Problem	Solution
1.1 Usable System	How can you design a system that is usable?	Understand the users and their use of the system.
2.1 Personas	Where do you start use-case modelling?	Start with the people who will actually use the system.
2.2 Candidate Use Case List	How do you determine what the system should do?	List **Candidate Use Cases** for each Persona.
2.3 Focal Use Cases	How can you manage a large number of candidate use cases?	Choose **focal** use cases to drive the design.
2.4 Use Case Diagram	How do you know when your candidate use case list is complete?	Draw a use case diagram to show how personas and use cases are related.
2.5 CRUD Use Cases	How do you get a complete set of use cases?	Apply CRUD analysis to each of the appropriate domain concepts.
2.6 Reporting Use Cases	How do you get a complete set of use cases for reporting?	Ensure you have at least twenty reporting use cases.
3.1 Use Case Dialogues	How can you describe what each use case involves.	Write essential use case dialogues for each use case.
3.2 Use Case Roleplay	How can you check that use case dialogues are correct?	Act each use case before an audience of the development team.
4.1 Modify State Use Case	How can the user get the system to do something that involves changing the state the system maintains?	Write a use case where the user provides information on the request, and the system has the responsibility for performing the command.
4.2 Query State Use Case	How can the user find something they need to know from the system?	Write a use case where the persona describes the information they require, and then the system presents that information.
4.3 Status Use Case	How can you let the user know a small amount of crucial information?	Write a use case where the system presents that information.
4.4 Alarm Use Case	How can the system inform the user about something?	Write a use case that begins with the system taking the responsibility to warn the user.
5.1 Prompting Step	How can you make sure the user has the information needed to make a decision?	Give the system the responsibility of offering that information before the user makes the decision.
5.2 Confirming Step	How should you ensure that correct information is communicated between the persona and the system?	Require the persona or system to confirm the information.
6.1 Extension	How do you model errors and exceptions in use cases?	Use extending use cases to record errors and exceptions.
6.2 Inclusion	How can you remove commonality between use cases?	Make a new use case containing the common steps, and include it in the use cases that have the common steps.
6.3 Specialisation	How can you handle different kinds of interactions that fulfil broadly similar goals?	Make more general and more specific use cases to capture the precise interactions.
6.4 Conditions	How do you model use cases than can only operate under certain circumstances?	Use pre- and post-conditions to control when use-cases are permissible.

Background

Jacobson et al. define a *use case* in their 1992 book to be "a behaviorally related sequence of transactions in a dialogue with the system" [Jacobson et al. 1992]. A more recent definition for the Rational Unified Process (RUP)

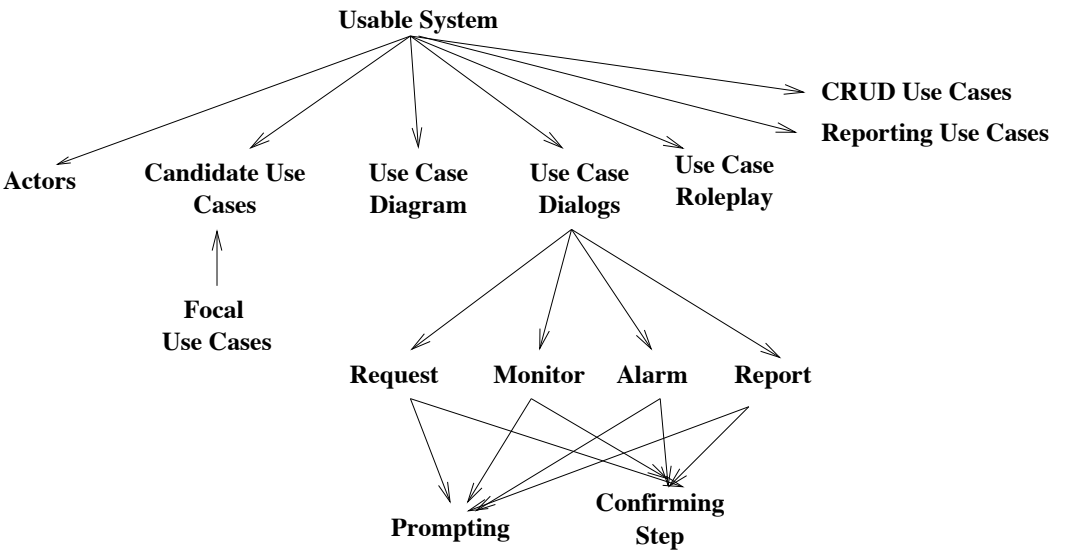

Fig. 1. The relationships between the patterns. The arrows show the *uses* relationship between patterns.

and the Unified Modelling Language (UML) shows little real change, saying a use case is "a description of a set or sequence of actions, including variants, that a system performs that yields an observable result of value to a particular persona" [Jacobson et al. 1999]. The general idea of a use case is to represent intended sequences of interaction between a system (even if not yet implemented) and the world outside that system.

Essential use cases were designed to avoid built-in assumptions, often hidden or implicit, about the form of the user interface expected by the RUP style use cases [Constantine and Lockwood 1999]. The term "essential" refers to essential models that "are intended to capture the essence of problems through technology-free, idealized, and abstract descriptions". Essential use cases are documented in a format representing a dialogue between the user and the system. This resembles a two-column format used by Wirfs-Brock [1993].

Form

The patterns are written in modified electric Portland form [Noble 2000]. 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. Some of the later patterns are presented in a condensed version of this style, where consequences are listed briefly rather than being discussed in depth: here + indicates a benefit (positive consequence of applying the pattern) and − indicates a liability (negative consequence).

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 management of service requests, and industry and academic courses.

1. CREATING A USABLE SYSTEM

The first pattern, **Usable System** provides a starting point for the patterns in this collection.

1.1 Usable System

How can you design a system that is usable?

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

—You are designing a system, or a system's user interface.
—Users can avoid using the system . . .
—but avoiding the system can impose significant costs on the business.
—Alternatively, users cannot avoid using the system, and so the system may impose significant costs on its users.
—You've a fixed time and cost budget for building the system.

It's one thing to want a system to be usable: it is another thing altogether to work to build a usable system. Bitter experience has shown that building a system and just hoping it will be usable is unlikely to result in a system that actually is usable.

So, how can we have the best chance to build a usable system?

Therefore: *Understand the users and their use of the system.*

Take the time to study the people who will use the system, the use they will make of it, and the context of that use.

We often define usability as "the fit between the person doing the work, the tasks required by the work, and the software to support the person doing the task". In terms of our patterns, the key force they address is usability, but usability has many facets — learnability, efficiency (that important tasks can be performed quickly), memorability, low error rates, likeability (that the user interface looks cool), and so on [Nielsen 1993]. Many of these patterns directly address some or all of these facets of usability — subforces — in the solutions and techniques that they provide.

More importantly, though, these patterns support the key dynamic of Usage-Centered Design: "*abstraction focused interaction*". Essential use cases are *essential* precisely because they focus only on the essentials of the interactions required for people to complete their work with the software. They elide any user-interface details, as described above, however they also elide any information about the skill of the user, the interaction media, nonfunctional constraints on their realisation, and so on. Omitting these crucial details *in the production of use cases* means that the use cases should focus solely on the interaction itself. The point, of course, is that use cases written in this way provide a sound foundation upon which interface design can build, explicitly addressing all these issues to produce highly usable computer systems.

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

Ideally, we would begin by observing the work of the potential users of the system, and then interviewing them in depth [Cooper et al. 2014]. We would no doubt also consult the client or project sponsor: after all they will be paying for the project. But to have any hope of ensuring usability, we must focus first on the users of the system and the tasks they will carry out, and understand both in detail.

Rather than giving all these details here, we will include them in subsequent patterns' examples, as they become relevant.

Consequences. By understanding the people who will use the system, the use they will make of it, and the context of that use, we are more likely to design usable systems. A usable system should encourage people to use it in lieu of other alternatives — and where those alternatives are e.g. phone helpdesks or in-person consultations, a computer system can answer questions at a tenth or less of the cost, and can be available at any time of the day or night. Where people are required to use a system by a business or a government, a more usable system will enable them to get their tasks finished more quickly, and can help mitigate health and safety concerns such as repetitive strain injury, eyestrain, or computer-related psychotic breaks.

However:

Starting by modelling personas and use cases may help result in a usable system, but may do little to placate stakeholders — or even developers, especially on a more aggressive Agile or DevOps project. Some kind of analysis of stakeholders goals and requirements, and on-going project-wide risk analysis, are important practices for all projects. However, these are separate activities from (and do not substitute for) analysing the goals and characteristics of the poor people who will actually have to put up with the system as part of their lives, and the uses to which they will put the system.

Discussion. 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.

Related Patterns. Constructing a **Usable System** should begin by determining the **Personas (2.1)** who will use the system. Then, personas' use should be captured as a set of **Candidate Use Cases (2.2)**, the **Focal Use Cases (2.3)** identified from within the candidates. A **Use Case Diagram (2.4)** can record relationships between use cases, such as **Extension (6.1)** to describe what happens when things go wrong, **Inclusion (6.2)** to factor our common interactions, and **Conditions (6.4)** to describe when things *shouldn't* happen Checking for **CRUD Use Cases (2.5)** and **Reporting Use Cases (2.6)** can help identify use cases that are often overlooked. The interactions for the focal use cases should be expanded into **Use Case Dialogues (3.1)**, and those dialogs validated in **Use Case Roleplays (3.2)**.

See Also. While aiming at usability, then, these patterns are primarily about analysing interaction. Many other pattern collections describe techniques for designing interfaces following an analysis [Borchers 2001; Tidwell 2002; Erickson 2002; Bradac and Fletcher 1988; Harrison et al. 2000].

2. FINDING USE CASES

Understanding how a system will be used is key to designing a usable system. For that reason, this section begins by modelling the people who will interact with the system — the *personas* — and then goes on to consider their patterns of use — the system's *use cases*. The first four patterns in this are about exploring the territory, making a rough map of the ground which you will cover in more detail using later patterns. These early 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.

2.1 Personas

Also Known As: Actors

Where do you start use-case modelling?

—You have to start modelling somewhere.
—There may be many stakeholders (other than users) 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" [Flight of the Conchords 2008]*) 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. A system may need to interface to fickle external systems; or meet strict technological or resource challenges [Noble and Weir 1998]. Given half a chance, stakeholders can easily booby-trap a development project before it gets started by insisting that time-to-market requirements preclude any consideration of usability.

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 **Personas** 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 personas in detail. For example, consider:

—personas' 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 personas**, that is, the important systems to which you have to interface. You should then characterise the system personas, describing their characteristics, typically by referring to existing manuals or protocol definitions.

Finally, you should roughly prioritise the personas 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 personas. 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 personas.

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 domain knowledge about the arts centre, especially when it comes to e.g. details of upcoming plays, subscription seasons, or membership discounts that they do not have to deal with every day. On the other hand, they are employees and so some minimal level of domain knowledge can be assumed (such as which plays are currently being performed) and also knowledge about the system (for example, through staff training). They will not be expected to know anything about computer systems in general, 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 Persona, but rather there is someone playing that role. For that reason, either this persona 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 persona** for the proposed system.

Looking over the list, we can see that it is already roughly 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 personas 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 personas 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 personas*, that is personas that use the system itself. Often there can be *indirect personas* 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 personas*, 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.

Related Patterns. Once we have identified a persona, we can characterise that persona's interactions with the system as **Candidate Use Cases (2.2)**.

2.2 Candidate Use Cases

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 which features would be nice to have but are not strictly necessary.
—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 personas doesn't help work out what a system should actually *do*. What you need to know is what the personas 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 Persona.*

Consider each Persona in turn, starting with the highest priority persona, and write a list of possible (candidate) use cases for each persona.

A use case is one complete case of system use. In other words, a use case describes a single sequence of interaction between a persona and a system. From the persona'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 personas) and try to understand them and their goals and intentions.

Example. Looking at the text and our list of personas, 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 4. 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 2.

Consequences. A list of use cases can give 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. Use cases 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.

Finding use cases should involve stakeholders 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). Involving users and stakeholders means it is usually fairly easy to quickly come up with a set of candidate use cases that are representative of what's actually wanted by the stakeholders, and what's actually needed by the users.

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.

However: identifying candidate use cases for personas 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 — especially if it turns out that their lists are wrong.

Related Patterns. To manage potentially large lists of use cases, and to help direct the design effort, we can identify a subset of the candidate use cases as **Focal Use Cases (2.3)**. Checking for **CRUD Use Cases (2.5)** and **Reporting Use Cases (2.6)** can help identify use cases that are often overlooked.

See Also. Use cases were first described by Jacobson for describing Danish telecommunications systems at Erickson [Jacobson et al. 1992]. There are number of other books describing use cases and their use in software development [Rosenberg and Scott 1999; Cockburn 2001; Armour and Miller 2001]. Other related patterns are listed in section 4.

2.3 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 personas are more important 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 [Orwell 1945]. They may take more time (or impose more risk) to implement, they may be more important to personas (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. Everyone 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, persona 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.

Related Patterns. It is generally worthwhile to expand out the interactions for focal use cases as **Use Case Dialogues (3.1)**, however less critical use cases can often simply left as a candidate name.

See Also. The Extreme Programming Planning Game [Beck and Fowler 2000] 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 no one represents the interests of the users — in the same way that no one 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.

2.4 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. 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 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?

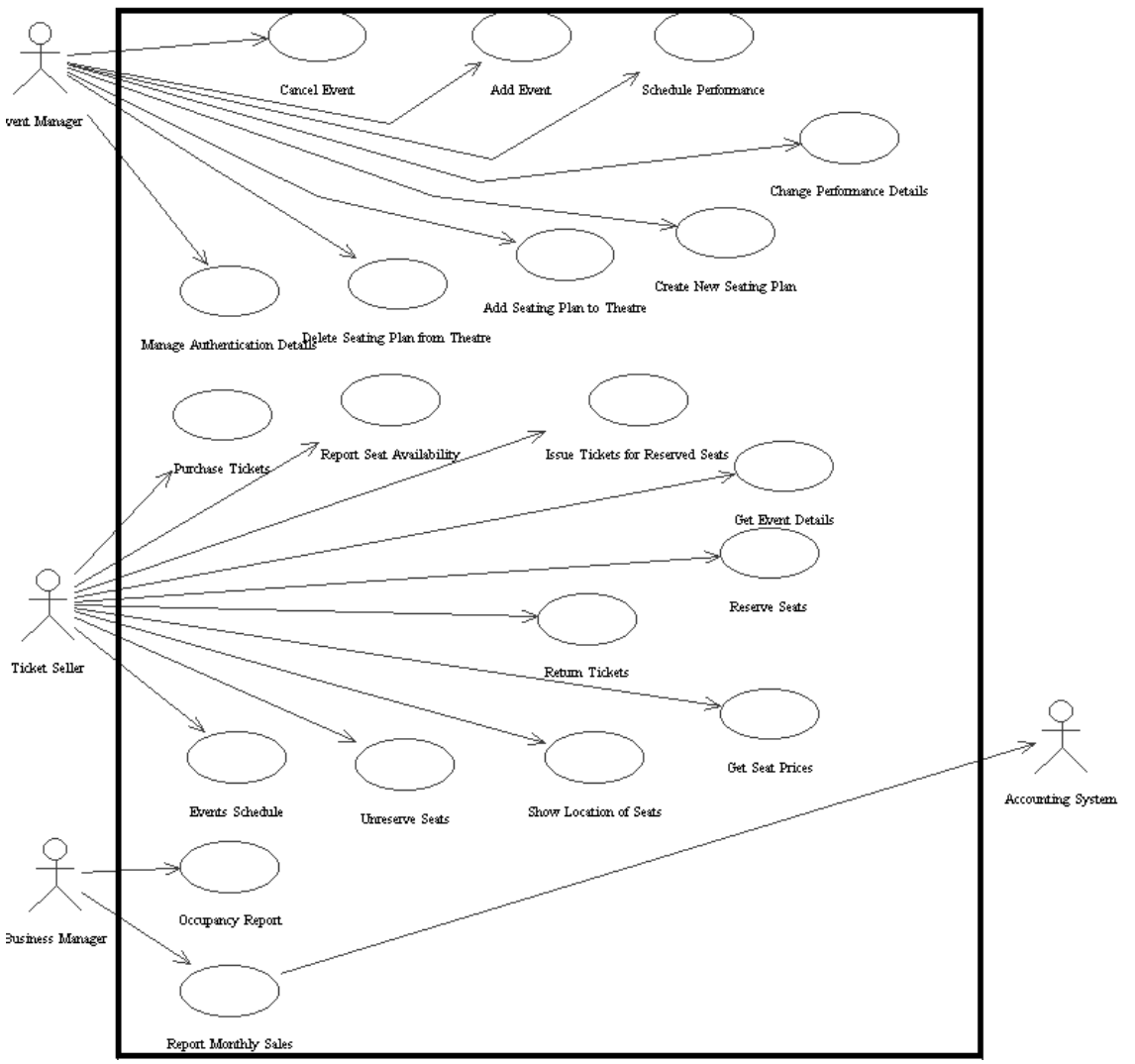

Fig. 2. A use case diagram, showing the system boundary.

Answers to these questions will need to convince other teams members, stakeholders, and so on, so how should you present your use case models to make these arguments?

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

A use case diagram shows stick figures for the personas 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 a persona representing every kind of user who will use the system?
—Is there a system persona for every external system with which this system needs to communicate?
—Can each persona 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 persona (or for a few related personas), rather than one diagram for the whole system.

Example. Figure 2 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 personas, 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.

Related Patterns. Inspecting Use Case Diagrams can help us identify missing **CRUD Use Cases (2.5)** and **Reporting Use Cases (2.6)**.

See Also. UML Distilled [Fowler and Scott 2000] briefly introduces use case diagrams. Software for Use describes more complex use case diagrams in more detail [Constantine and Lockwood 1999]. Jacobson et al. use a system box [Jacobson et al. 1992].

2.5 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 that people 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 a 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.

Related Patterns. **Reporting Use Cases (2.6)** describes how analysing reporting requirement can also help identify missing use cases.

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 [Cockburn 2001]. 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.

2.6 Reporting Use Cases

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

Even though reporting is boring, if it's important to someone, you have to think about it. Given that many systems collect, accumulate, and warehouse data, it is just as important to get that data out of the system as it is to get the data in.

—Reporting is very important for lots of systems
—Reporting is boring for implementers, 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 personas

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

Analyse the problem, domain model, talk to users, brainstorm, 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 2 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 being forced 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.

Related Patterns. Checking for **CRUD Use Cases (2.5)** describes how analysing data flows can also help identify missing use cases.

3. 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 coming up with the "whole" use case model.

3.1 Essential Use Case Dialogues

Also Known As: Use Case Bodies, Use Case Cards

How can you describe what each use case involves?

—You have a prioritised list of personas — 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 (2.2)**, identified the **Focal Use Cases (2.3)**, and perhaps checked some **Use Case Diagrams (2.4)**, you still don't really have much detail about what the use cases will involve: what information needs to be provided by personas, and how the system should behave to implement the use cases.

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 will probably be 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

gettingCash *(conventional)*

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	

gettingCash *(essential)*

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

Fig. 3. On the left, a conventional use case for getting cash from an automatic teller system. On the right, an essential use case for the same interaction. (From Constantine and Lockwood.)

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 [1999]. 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 example shown in figure 3. The dialogue on the left of figure 3 is for a conventional use case, described in terms of actions and responses. The dialogue on the right of figure 3 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.

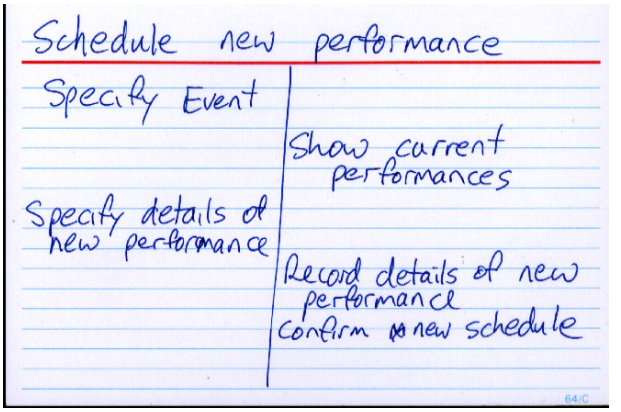

Fig. 4. A use case card showing the essential use case details for Schedule new performance.

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 persona has already chosen to do this use case and has already told the system that they are doing it, so we don't need to include a separate step to start the use case.

Example. Figure 4 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 [Jacobson et al. 1999]).

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 dialogues of each use case — if two dialogues are the the same, then they should probably be the same use case, so eliminate one of them. If one case seems to need more than one dialogue, you probably need different use cases. Two use cases that are similar can be modelled by **Specialisation (6.3)** or **Inclusion (6.2)**. Similarly, errors that can occur during a use case can be modelled by **Extensions (6.1)** or **Conditions (6.4)**.

Use case cards were inspired by Class-Responsibility-Collaborator (CRC) cards [Beck and Cunningham 1989; Wilkinson 1996; Bellin and Suchman Simone 1997]. They are also similar to the Story Cards used to schedule Extreme Programming iterations [Beck and Fowler 2000]. Wirfs-Brock introduced the idea of the two-column format [Wirfs-Brock 1993].

3.2 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.
—Team members need 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 a persona 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 adds more fun and variety into the activity, and these also heighten attention.

See Also. Use case roleplays were inspired by CRC Card Roleplays [Wilkinson 1996; Bellin and Suchman Simone 1997]. Further discussion on use case roleplay can be found in previous work [Biddle et al. 2001b]. Our use of roleplay is similar to Wirfs-Brock's use of "conversations" to evaluate use cases [Wirfs-Brock 1994].

4. USE CASE DIALOG PATTERNS

Once you start writing use cases you'll realise that lots of kinds of use cases come up over and over again — that there are actually *patterns* in the dialogues of essential use case themselves. These patterns typically support repeated stereotyped interactions, such as displaying or modifying system state, reporting system status, or raising alarms.

This section lists a number of these patterns. In the interests of space, we give only the bare bones of each pattern.

4.1 Modify State Use Case

How can the user get the system to do something that involves changing the state the system maintains?

—Sometimes the user needs to get the system to do something.
—That something requires the system to change its state.
—Exactly what changes take place depends on information that the user has.

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

This is the simplest and most common kind of use case: the user's commands are listed in the left-hand column, and the system's responsibility in the right-hand column. Often this kind of use case can require just one user intention step, and one system responsibility step, however more complex interactions, requiring more information can have many steps for both the user and the system.

Example. An important use case for the arts centre system is the provide tickets for seats for a performance.

Purchase Tickets

User Intention	System Responsibility
Choose performance and seats	
	Record seats at performance as reserved
	Print tickets

Consequences

+ The system executes the user's command.

Discussion. Typically use cases should only consider the *"happy path"* — that is, you should assume the use case is only ever started when it makes sense, and that the use case always executes correctly (this term is due to Rebecca Wirfs-Brock [1993]). You should use **Extension (6.1)** to describe what happens when things go wrong, and **Conditions (6.4)** to describe when things *shouldn't* happen.

Related Patterns. If the command is important, you may need to include a **Confirming Step (5.2)**:

Print performance schedule

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

This allows the user to know the command has been completed, even if it is not immediately obvious.

4.2 Query State Use Case

How can the user find something they need to know from the system?

—Sometimes the system knows things that users don't know, and users need to find this out.
—The information may be simple, but there may be large amounts of information
—Some information may be more important than other information
—Users have their own idiosyncratic ways of thinking about the world, which may not match the way you (or your systems analysts and designers) think about it.

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

Example

Get Seat Prices

User Intention	System Responsibility
Choose performance	
	Show prices for chosen performance

The user chooses the theatre performance they wish to view, then the system will provide seat prices for that performance. There can be easily up to ten price classes for any performance, so all the prices cannot be displayed for all the performances on a single screen.

Consequences

+ The user can get just the information that they need.
+ If the information is large or complex, they can specify just that information they need to see.
- The user still has to describe what they need accurately enough for the system to know what is wanted.

Related Patterns. The use case can also involve a **Prompting Step (5.1)**. If only a small amount of important information needs to be available constantly, consider a **Status Use Case (4.3)**.

4.3 Status Use Case

How can you let the user know a small amount of crucial information?

—Some information the system has is more important than other information.
—Some information may change frequently, or asynchronously with respect to the user.
—Some information may be important to the continued use of the system — for example, some other use cases may (or may not) be permitted only in certain system states.

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

Example

Show Next Performances

User Intention	System Responsibility
	Continuously show the remaining performances today

This is a minimal use case: it simply provides (generally a small amount of) important information to the user. In a realisation of the system with a graphical user interface, this information is typically displayed in a status bar or on a window background.

Consequences

+ The status information is constantly available.
- The status monitors constantly take up display real estate.
- Changes to the status information can distract the user from more important tasks.

Related Patterns. If changes to the information are more important than actual values (or of some values are more important than others) then consider an **Alarm Use Case (4.4)** as an alternative.

To report larger amounts of information, consider the **Query State Use Case (4.2)** and **Reporting Use Cases (2.6)** as alternatives.

4.4 Alarm Use Case

How can the system inform the user about something?

—The system needs to draw the persona'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, personas 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, theatre, and times of performance

Essential use cases generally begin with a left-hand-side user intention — see e.g. Figure 3 and most of the other use cases in this paper. In contrast, Alarm Use Cases start in the right-hand-side column, with a system responsibility. That is, it's the systems job to "start" the use case, not the user's.

Consequences

+ The system takes responsibility for initiating the use case.
+ The system can pass information about the alarm to the persona.
+ The persona does not have to interrupt their current task immediately to respond to the alarm.
- The persona can ignore the alarm.
- The persona no longer initiates action, and so loses control over the system.

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.

Discussion. If the alarm is important, you may need to include a **Confirming Step (5.2)**:

Warn theatre performance undersold

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

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

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

5. USE CASE STEPS

As well as patterns of whole use cases, there are also patterns *within* the dialogues of individual use cases. This section describes the two main patterns we have seen within use cases — prompting for a new interaction and confirming a previous interaction.

5.1 Prompting Step

How can you make sure the user has the information needed to make a decision?

—Sometimes users need to make decisions based on information that is held by the system.
—This information may or may not be known by users.
—It's easier for people to choose between several options in front of them than it is to remember what options are possible.

Therefore: *Give the system the responsibility of offering that information before the user makes the decision. Whenever your use cases require input from the user — consider if the system knows the most likely or most common inputs. If so, prompt the users with these common inputs before they make their decision.*

Example

Reserve seats for performance

User Intention	System Responsibility
	Offer unreserved seats
Choose seats	
	Record seats reserved

Consequences

+ Users have the information they need before they make the decisions.
+ Users are more likely to make correct decisions, decreasing the error rate.
+ Because information is presented to them, users do not need to remember it, increasing learnability.
- The information you supply could bias the user's choices.

- Prompts can require screen real estate (or other interface resources), obscuring other information that is actually more important to users.

5.2 Confirming Step

How should you ensure that correct information is communicated between the persona and the system?

—Some information is more important than other information.
—When important information is communicated, it can be very important to ensure it is communicated accurately.
—Similarly, some commands are sufficiently important (or dangerous) that they should only be performed correctly.

Therefore: *Require the persona 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

Consequences

+ Users have an opportunity to confirm their data or actions have been correctly interpreted by the system (and vice versa).
- Confirmations can require screen real estate (or other interface resources), obscuring other information that is actually more important to users.
- Confirmations can distract users from more important tasks.
- Familiarity breeds contempt: confirmations can easily become routine. If users consider them part of their everyday operation of the system (to book a ticket, click "book", "confirm", "confirm") then every command will be confirmed instinctively, even if erroneous.

Discussion. The insertion of a confirming step (either as the user's intention or the system's responsibility) does not necessarily require the traditional dialog-box style implementation. The problems with such confirmations are well known. Tog has described how wait timeouts can be used to confirm actions, rather than dialog boxes: in these cases, the user does *nothing* to confirm a correct operation but has a few seconds grace to abort an incorrect operation [Tognazzini 1998].

6. ORGANISING USE CASES

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

6.1 Extension

How do you model errors and exceptions in use cases?

—You write essential use cases to described the "happy path" — that is describing the correct behaviour of the system.
—In the real world things go wrong: you have to deal with the "unhappy path" somehow.
—Every use case makes the description of the system more complex (and probably the system as a whole more complex too).

Therefore: *Use extending use cases to record errors and exceptions.*

One use case (called the *extending case* or *extension*) can *extend* another use case (called the *base case*). This means that whenever the base case is enacted, execution can switch to the extension case instead. If the extension completes successfully, the base case can continue; otherwise, the extension terminates the execution of the base case.

Example. Consider again the "Pay for reservation" use case (§5.2).

Pay for reservation

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

This is written following the happy path: it assumes that the payment is successful. We could write an extending use case to describe what happens when the payment is declined (when the Validate payment step fails).

Payment declined

extends: Pay for reservation

User Intention	System Responsibility
	Show payment failure
	Offer alternative payment methods
Choose payment method	
Supply payment details	
Confirm method and details	
	Accept payment

In this case, the user is offered a choice of an alternative payment method; if this fails in turn, then the whole Pay for reservation use case has failed.

Consequences

+ Extensions allow you to describe errors or exceptions.
+ Extensions ensure that these descriptions do not clutter the body of the happy path use case.
- When designing or implementing a use case, you have find (and then consider) all the use cases that extend it.
- You can easily overuse extensions. You should assume interface and software designers will make common-sense decisions in situations in common-sense situations, and concentrate on describing the important, strange, interesting, or weird.

Discussion. The need for an extension use case is often indicated by phrases in a step that indicate that the step will not always complete successfully, such as "check" or "validate".

Sometimes you need to have a use case that extends every other use case — consider e.g. a pervasive system error such as e.g. a network link going down, or simply the requirement that users can terminate interactions with the system at any time. You can identify these use cases by annotating them "extends *".

6.2 Inclusion

How can you remove commonality between use cases?

—Many use cases can contain common steps.
—Repeating these common steps is not only boring: it also creates consistency problems.

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

One use case (called the *included case* or *inclusion*) can be *included* in another use case (called the *base case*). In the body of the base use case, you can write "> anotherUseCase" to include the steps of the included case — this can appear on either column of the base case. Whenever that step of the base case is enacted, you execute the included use case; when the included case is complete, you continue with the next step of the base use case.

Example

Print tickets

User Intention	System Responsibility
	Print ticket for specified seats
	Record tickets printed
	Update ticketed printed count

Purchase tickets

User Intention	System Responsibility
Specify performance and seats	
	>**Print Tickets**

Issue tickets for reserved seats

User Intention	System Responsibility
Identify reserved seats	
	>**Print Tickets**

Consequences

+ Common steps are localised in a single use case.
+/- Changing an included use case changes all the use cases into which it is included.
- You have to understand all the subsidiary use cases to understand a use case that includes them.
- If you're not careful, you can end up doing procedural design with use cases as procedures and inclusions as subroutine calls. The aim is to capture patterns of use, not not to design program logic via structured flowcharts. Keeping use cases independent or linked via extensions makes clear that the the order of interactions should be flexible where possible.

6.3 Specialisation

How can you handle different kinds of interactions that fulfil broadly similar goals?

—Sometimes there are many similar tasks users need to do.
—Some ways of using one use case may be more common or more important than other ways of using the same case.
—User interfaces need to recognise commonality between similar use cases to provide consistent designs; however they also need to recognise frequent use cases to provide efficient designs.

Therefore: *Make more general and more specific use cases to capture the precise interactions users will have with the system.*

One use case (called the *special case*, *subcase*, or *specialisation*) can *specialise* another use case (called the *base case*, *general case*, or *supercase*).

Unlike extends or uses, this relationship puts an informal requirement on the dialogues of the base and special cases: the special case should be a "special version of" the base case. The relationship also means that whenever the supercase may be enacted, the subcase may be enacted instead.

Example

Print performance schedule

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

A particularly common version of this use case will be to print the schedule for just today. Since it is probably worth optimising the user interface design and/or the software design to support this particular version, we can make a specialisation of the base case to handle this situation.

Print today's performance schedule

specialises: Print performance schedule

User Intention	System Responsibility
	Print schedule of performances for today, and for the next day with performances

Consequences

+ You can group your use cases into categories or hierarchies.
+ You can identify important special versions of more general use cases.
- Specialisations can make use case models more complex.

Related Patterns. If you seem to have a choice between **Inclusion (6.2)** and specialisation, choose inclusion, just as you would prefer composition to inheritance in other object-oriented models and designs.

6.4 Conditions

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

—Some use cases can only be enacted at certain times or in certain situations.
—Some use cases can only be enacted in under certain conditions or states of the system or the world.
—Some use cases can only be enacted after other use cases have been enacted.

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

A *precondition* is something that must be true before a use case can be enacted; a *postcondition* is something which is true afterwards. If a use case has a precondition, that condition can be assumed to be true before the use case is executed; if it has a postcondition, that condition can be assumed to be true afterwords.

Example. The theatre booking system could require users to log in before they can use the system. We can model this with a condition called "User is logged in".

Login

User Intention	System Responsibility
Precondition: not the current user is logged in	
Identify self	
	Verify identity
	Postcondition: the current user is logged in

Then, every use case could require the user to be logged in:

Print performance schedule

User Intention	System Responsibility
Precondition: the current user is logged in	
Chose start and end dates	
	Print schedule of performances from start to end date

Finally, users should log out at the end of the session.

Logout

User Intention	System Responsibility
Precondition: the current user is logged in	
	Confirm logout
	Postcondition: not the current user is logged in

Extra for experts: the system could log a user out if they have been idle for say 30 minutes:

Timeout

extends: *

User Intention	System Responsibility
Precondition: the current user is logged in	
	Wait until user has been idle 30 minutes
	> Logout

We've described this here using extensions, inclusions, and conditions. This is really a cautionary example: you should never need anything so baroque in practice.

Consequences

+ Conditions can make it clearer when particular use cases can execute.
- Conditions can make use case models much more complex. You rarely need more than two conditions in any application.
- In particular, resist the temptation to construct complex application-level state machines in use case conditions.

Discussion. There are several useful kinds of conditions.

—the value of an imaginary, application-global boolean variable (e.g. "the current user is logged in", "networkIsConnected"),
—the fact that some other use case has been executed in this session (e.g. "User login"),
—a description of a condition in the system or the real world (e.g., "Identity has been verified"), or
—the negation of some other condition.

Note that the login example could be simplified by using an implicit "Login" use case name condition instead of a special "the current user is logged in" condition. Doing it implicitly is a little more terse but means exactly the same as the explicit conditions in the example above. The only differences is that the Login use case would not need to establish a postcondition, because the "Login" condition is established automatically when its homonomous use case completes successfully.

Note also that *including* "Login" in another use case is different from having "the current user is logged in" as a pre-condition. The latter only requires that the user be logged in in order to carry out the use case. The former requires that the user logs in on every enactment of that use case ("Authenticate" would be a better name in that case).

Pre- and post-conditions should match up in a complete model: that is, for any condition, there should be at least one use case that establishes it as a postcondition, at least one use case that depends upon it as a precondition, and at least one use case that cancels it (that is, its postcondition is the negation of the condition).

Related Patterns. If you seem to have a choice between **Extension (6.1)** and conditions, choose extensions.

DISCUSSION AND CONCLUSION

In this paper, we have presented a number of patterns for beginning to address the usability of a system by applying usage-centred design, in particular using essential use cases. We hope these patterns provide concrete guidance to developers and designers, to help them manage stakeholders, to identify users and their requirements, to prioritize requirements between developers, stakeholders, and users, to find a "way in" to a system at the start of a project — and, given a modicum of luck and good judgement, to produce a more usable system in the end.

Many of these patterns may also be applicable to conventional use cases, or to other forms of user and task modelling such as scenarios [Cooper et al. 2014] or user stories [Beck 1999] — although we believe the patterns are more evident in the essential use cases than in other modelling techniques. Cooper's scenarios can be abstracted into essential use cases, and essential use cases can be detailed to produce Cooper-style narrative scenarios, but it is not clear how these techniques can best be applied together.

We expect to improve upon the patterns we have discussed here, and we are investigating other possible patterns. These patterns stop at modelling: a key advantage of Constantine & Lockwood's approach is that an information architecture or interaction framework can be derived directly from a use-case model. Clearly there is an opportunity for more patterns here.

In the future, we hope to address other aspects of usability, such as how best to characterise personas / actors / user roles: via a textual scenario, a set of facets and attribute values, or some combination? We would like to incorporate techniques to help ensure a design is not biased e.g. against particular kind of users based on their cognitive style [Hill et al. 2017], or to consider users' emotional responses to interfaces [GhasemAghaei et al. 2017].

We would also eventually like to address questions about development processes, such as deciding when to switch between planning and implementing, or how designers can build models piece-wise in iteration-based development approaches [Ferreira et al. 2007; Brown et al. 2012].

ACKNOWLEDGEMENTS

Thanks to Joe Corneli, the PLOP 2017 shephered for this paper, for his detailed and insightful comments, and even more for being patient as we shipped him drafts at the last possible minute. Thanks also to the participants from KoalaPLoP 2001 Workshop C — Saluka Kodiuwakku, Pauline Khoo, and John Hosking — and the KoalaPLoP 2002 Workshop — Pronab Ganguly, Pradip Sarkar, Marie-Christine Fauvet, and Brian Foote. — for their comments on versions of earlier papers. Finally, thanks are due to Larry Constantine and Lucy Lockwood for describing usage-centred design in the first place.

REFERENCES

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

Kent Beck. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Kent Beck and Ward Cunningham. 1989. A Laboratory for Teaching Object-Oriented Thinking. In *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*. 1–6.

Kent Beck and Martin Fowler. 2000. *Planning Extreme Programming*. Addison-Wesley.

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

Robert Biddle, James Noble, and Ewan Tempero. 2001a. Patterns for Essential Use Cases. In *KoalaPLoP 2001: The Second Asian-Pacific Conference of Pattern Languages of Program Design*, James Noble and Brian Wallis (Eds.). Melbourne, Australia.

Robert Biddle, James Noble, and Ewan Tempero. 2001b. Role-play and Use Case Cards for Requirements Review. In *Twelfth Australasian Conference on Information Systems*. Coffs Harbour, Australia, 13–22.

Robert Biddle, James Noble, and Ewan Tempero. 2002. Patterns for Essential Use Case Bodies. In *KoalaPLoP 2002: The Third Asian-Pacific Conference of Pattern Languages of Program Design (Conferences in Research and Practice in Information Technology)*, James Noble and Paul Taylor (Eds.). Melbourne, Australia.

Jan Borchers. 2001. *A Pattern Approach to Interaction Design*. John Wiley & Sons, Ltd.

Marc Bradac and Becky Fletcher. 1988. A Pattern Language for Developing Form Style Windows. In *Pattern Languages of Program Design*, Robert Martin, Dirk Riehle, and Frank Buschmann (Eds.). Vol. 3. Addison-Wesley, 347–358.

Judith M. Brown, Gitte Lindgaard, and Robert Biddle. 2012. Interactional identity: designers and developers making joint work meaningful and effective. In *CSCW '12 Computer Supported Cooperative Work, Seattle, WA, USA, February 11-15, 2012*. 1381–1390.

Alistair Cockburn. 2001. *Writing effective use cases*. Addison-Wesley.

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

Alan Cooper, Robert Reimann, David Cronin, and Christopher Noessel. 2014. *About Face: The Essentials of Interaction Design* (4th ed.). Wiley.

Tom Erickson. 2002. Interaction Design Patterns Page. `http://www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html`. (2002).

Jennifer Ferreira, James Noble, and Robert Biddle. 2007. Agile Development Iterations and UI Design. In *AGILE 2007 Conference (AGILE 2007), 13-17 August 2007, Washington, DC, USA*. 50–58.

Flight of the Conchords. 2008. Robots. (2008).

Martin Fowler and Kendall Scott. 2000. *UML Distilled: A brief guide to the standard object modeling language* (second ed.). Addison-Wesley.

Reza GhasemAghaei, Ali Arya, and Robert Biddle. 2017. Affective Walkthroughs and Heuristics: Evaluating Minecraft Hour of Code. In *Learning and Collaboration Technologies. Technology in Education - 4th International Conference, LCT 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings, Part II*. 22–40.

Neil Harrison, Brian Foote, and Hans Rohnert (Eds.). 2000. *Pattern Languages of Program Design*. Vol. 4. Addison-Wesley, Chapter Part 7: Patterns of Human-Computer Interaction, 445–593.

Charles Hill, Maren Haag, Alannah Oleson, Chris Mendez, Nicola Marsden, Anita Sarma, and Margaret Burnett. 2017. Gender-Inclusiveness Personas vs. Stereotyping: Can we have it both ways?. In *ACM Conference in Human Factors in Computing Systems (CHI)*.

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

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

Jakob Nielsen. 1993. *Usability Engineering*. Academic Press.

James Noble. 1998. Classifying Relationships between object-oriented design patterns. In *Proceedings of the 1998 Australian Software Engineering Conference*, Douglas D. Grant (Ed.). 98–109.

James Noble. 2000. Arguments and Results. *Comput. J.* 46, 3 (2000), 439=450.

James Noble and Charles Weir. 1998. Proceedings of the Memory Preservation Society. In *EuroPLoP*. 367–400.

George Orwell. 1945. *Animal Farm*. Secker and Warburg.

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

Jenifer Tidwell. 2002. UI Patterns and Techniques. `http://time-tripper.com/uipatterns/`. (2002).

Bruce Tognazzini. 1998. The Polite Interface or Guidelines for Dialogs. "Ask Tog" http://www.asktog.com/columns/012dialogGuidelines.html. (Sept. 1998).

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

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

Rebecca J. Wirfs-Brock. 1993. Designing Scenarios: Making the Case for a Use Case Framework. *The Smalltalk Report* 3, 3 (1993).

Rebecca J. Wirfs-Brock. 1994. The Art of Meaningful Conversations. *The Smalltalk Report* 4, 5 (1994).

PLoP'17, OCTOBER 23–25, Vancouver, British Columbia, Canada. Copyright 2017 is held by the authors. HILLSIDE 978-1-941652-03-9