Patterns for Essential Use Case Bodies

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

March 13, 2002

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 contains patterns that describe how to write the *bodies* of essential use cases. Readers of this paper will be able to write good 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 chicklet 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.*

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* [3], Usage Centred Design is based on *essential use cases*, and draws from ideas from object-oriented methodology [4, 5, 2, 1] 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 extends our earlier collection, *Patterns for Essential Use Cases* [**?**] to focus on the mechanics of writing the bodies of

individual use cases and the relationships between them — Figure 1 summarises the problems dealt with by this collection of patterns, and the solutions they provide. Larger-scale process issues such as what essential use cases actually are, why you might want to find them, and how to test them are covered in the earlier collection.

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. We hope these patterns will complete the large "literature" about use cases frantically being published by many fine international publisers. Essential use cases are design to be short, simple, quick, and to the point: we hope these patterns share these virtues.

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 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 programming logic controllers), projects for managing telecommunications plant and managment of service requests, and industry and academic courses and case studies.

Pattern	Problem	Solution
Commanding Use Case	How can the user get the system to do something?	Write a use case where the user provides information on the request, and the system has the responsibility for performing the command.
Requesting Use Case	How can the user find someting they need to know from the system?	Write a use case where the actor describes the information they require, and then the system presents that information.
Monitoring Use Case	How can you let the user know about a relatively small amount of important information from the system.	Write a use case where the system presents that information.
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.
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.
Confirming Step	How should you ensure that correct information is communicated between the actor and the system?	Require the actor or system to confirm the information.
Extension	How do you model errors and exceptions in use cases?	Use extending use cases to record errors and exceptions.
Inclusion	How do 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.
Specialisation	How can you handle different kinds of interactions that fullfil broadly similar goals?	Make more general and more specific use cases to capture the precise interactions.
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.

Figure 1: Summary of the Patterns

1 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.

1.1 Commanding Use Case

How can the user get the system to do something?

- Sometimes the user needs to 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.*

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 infomation can have many steps for both the user and the system.

Note that essential use cases do not have an explicit step to choose the execution of this use cases as against any other use case. Rather, essential use cases are written under the assumption that the user interface design has already indicated that this use case should be carried out: how that choice is made is purely an issue for the user interface. In some cases, an explicit interface interface element may be required (a voice menu entry or GUI command button), but in other designs may determine this implicitly (say by dragging the mouse).

Example

A simple use case for the arts centre system is to print out a performance schedule, so that potential clients can inspect it at their leisure (and hopefully return to make a booking).

Print performance schedule

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

Consequences

+ The system executes the user's command.

Discussion

Note also that use cases should only consider the "happy path" — that is, they should be written under the assumption that it makes the use case is only every started when it makes sense to do that use case, and that the case always executes correctly (this term is due to Rebecca Wirfs-Brock [**?**]). You should use **Extension (2.1)** and **Conditions (2.4)** to describe what happens when things go wrong.

Related Patterns

If the command is important, you may need to include a **Confirming Step (1.6)**:

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.

1.2 Requesting Use Case

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

- Sometimes the system knows things that users doesn't know, and users need to find this out.
- The information may be simple, but there may be large amounts of information (much of which is not of interest).
- Users have their own idiosyncratic ways of thinking about the world, which may not match the way you (or your systems analysts and designers) thing about it.

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

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 display 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
- An extra user step is required to choose that information.

Related Patterns

This use case can also involve a **Prompting Step (1.5)**. In fact, the example above does this, the system prompts to "Offer performances" before the user "Choose performance". If there a small amount of important information needs to be available constantly, consider a **Monitoring Use Case (1.3)**.

1.3 Monitoring Use Case

How can you let the user know about a relatively small amount of important information from the system.

- Some important the system has is more important than other information.
- Some information may change frequently, or asychronously 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 certainly system states.

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

Example

Show Today's Performances

User Intention	System Responsibility
	Show today's performances

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 instanteanous changes to the information, is more important the value of the information (or some values are more important than others) then consider an **Alarm Use Case (1.4)** as an alternative.

1.4 Alarm Use Case

How can 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

The key point of this use case is that it starts in the right-hand-side column, with a system responsibility (whereas use cases generally being with a left-hand-side user intention). 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 actor.
+ The actor does not have to interrupt their current task immediately to respond to the alarm.
- The actor can 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.

Discussion

If the alarm is important, you may need to include a **Confirming Step (1.6)**:

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:

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

1.5 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	

Consequences

+ Users have the information they need before they make the decisions
- 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.

1.6 Confirming Step

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

- Some information is more important than other information.
- When important information is commmunicated, 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 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

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 instincitevly, even if erroneous.

Discussion

Once again, it is important to note that the insertion of a confirming step (either as the user's intention or the system's responsibility) to does not necessarily require the traditional dialog-box style implementation, especially as the problems with such confirmations are well known. For example, 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 [**?**].

2 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.

2.1 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 contine; or the extension can terminte the execution of the base case.

Example

Consider again the "Pay for reservation" use case (§1.6).

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

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 Accept 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

Sometimes you need to have a use case that extends every other use case: you can write this as "extends *".

2.2 Inclusion

How do 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 contine with the next step of the base use case.

Example

needs to be done — can't think of a good one now.

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!

2.3 Specialisation

How can you handle different kinds of interactions that fullfil 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 bodies 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 encated instead.

Example

Consider again the Print performance schedule use case from (§1.1):

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 and tomorrow. 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 (2.2)** and specialisation, choose inclusion.

2.4 Conditions

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

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

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

A *precondition* is something which 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 afterwards.

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 User is logged in	
Identify self	
	Verify identity
	Postcondition: User is logged in

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

Print performance schedule

User Intention	System Responsibility
Precondition: 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: User is logged in	
	Confirm logout
	Postcondition: not 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: 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 — list the name of the variable (e.g. "User is logged in", "networkIsConnected")
- that some other use case has been executed in this session — list the name of the use case (e.g. "User login"). That is, each use case automatically sets one of the imaginary global variables after they execute.
- a description of a condition in the system or the real world
- **not** some other condition

Note that the login example could be simplified by using an implicit "Login" use case name condition instead of a special "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" econdition is established automatically when its homonomous use case completes successfully.

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 (as a **not**postcondition).

Related Patterns

If you seem to have a choice between **Extension (2.1)** and conditions, choose extensions.

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.

References

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

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

[3] 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.

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

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