VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

Supporting Reusable Use Cases

Robert Biddle, James Noble, and Ewan Tempero

Technical Report CS-TR-01/10
October 2001

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Supporting Reusable Use Cases

Robert Biddle, James Noble, and Ewan Tempero

Technical Report CS-TR-01/10
October 2001

Abstract

Use cases are common practice for capturing functional requirements for object-oriented software development. But, as with any artifact, there is a cost of producing them. Reusing use cases can reduce this cost. We discuss how use cases can be made reusable, in particular looking at the reusability of *essential use cases*. We also introduce Ukase, a web-based use case management tool that provides support for the reuse of use cases.

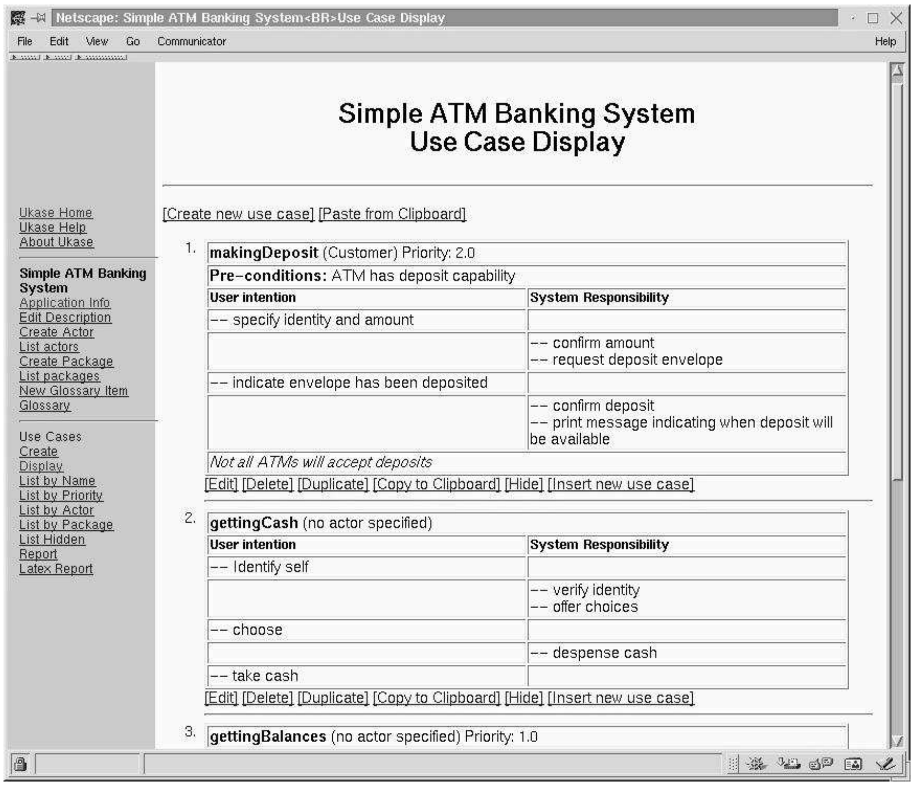


Figure 1: Ukase is a web-based tool for managing essential use cases.

1 Introduction

Use cases [11] are common practice for capturing functional requirements for object-oriented software development, and they are widely supported in modelling languages such as UML [19, 9], and in development processes such as the Rational Unified Process [10]. Given the investment organisations are making in use cases, it is reasonable to seek ways to reduce the cost of their development. One possibility is to *reuse* use cases, but in order for this to be effective, we need to understand what makes use cases *reusable*. In this paper, we examine the concept of reusable use cases.

One issue that affects the likelihood that anything will be reused is the form in which it is described. We have been using a particular form of use cases, known as *essential use cases*. These were introduced by Constantine and Lockwood for their *Usage-Centered Design* [6]. Essential use cases are written specially to be abstract, lightweight, and technology-free. Our observation is that use cases written this way are applicable in more situations than the conventional forms of use cases, which suggests they are more reusable. In order to gain an objective understanding of their reusability, we apply the reusability support model [3]. This model suggests the kind of support for reuse that is appropriate. We use this to develop tool support in the form of Ukase, a web-based use case management tool (see figure 1).

Our paper is organised as follows. In the next section, we discuss use cases and essential use cases, and summarise the reusability support model. Section 3 examines reusability of use cases by traversing the model. We then introduce Ukase and examine how reuse of use cases is supported in section 4. Section 5 discusses related work, and we present our conclusions in section 6.

2 Background

2.1 Use Cases

Jacobson et al. define a use case in their 1992 book as "a behaviourally related sequence of transactions in a dialogue with the system"[11]. A more recent definition for the Rational Unified Process 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 actor"[10].

The general idea of use cases is the representation of intended sequences of interaction between a system (even if not yet implemented) and the world outside that system. This idea is very powerful, for several reasons.

In the early stages of development, use cases help to focus on interactions as a way of eliciting desirable system behaviour, and so help capture requirements and determine specifications. The narrative or dialogue form typical of use cases is very easy for people to recall or imagine, and especially useful when involving end-users, background stake-holders, and others with no direct experience or role in actual system development. In the later stages of development, use cases help again because of the focus on interactions. The interactions can now be regarded as the embodiment of functional specifications that the system must meet.

Use cases also lead to a useful partitioning of requirements. This happens naturally, because use cases are based on sequences of interaction, and desirable interactions typically follow a structure of coherent progression, on a limited scale, toward a goal or sub-goal. This partitioning then allows organisation by grouping, filtering, prioritising, and so on, and is helpful in overall management throughout development.

2.2 Essential Use Cases

Essential use cases are part of Usage-Centered Design [6], developed by Larry Constantine and Lucy Lockwood, who see some difficulties with ordinary use cases. "In particular, conventional use cases typically contain too many built-in assumptions, often hidden or implicit, about the form of the user interface that is yet to be designed."

Essential use cases were designed to overcome these problems. The term "essential" refers to essential models that involve "technology-free, idealised, 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 [23]. In Wirfs-Brock's format, the column labels refer to the *action* and the *response*. In contrast, essential use cases label the columns as *user intention* and *system responsibility*.

These new labels indicate how essential use cases support abstraction by allowing the interaction to be documented without describing the details of the user interface. Note that the abstraction does not really relate to the use case as a whole, but more to the steps of the use case. In this way an essential use case does specify a sequence of interaction, but a sequence with abstract steps. Constantine and Lockwood give the examples shown in figure 2. The steps of the essential use case are more abstract, and the whole use case is more brief.

Constantine and Lockwood introduced essential use cases for user interface design and development. We have, however, applied essential use cases in more general systems analysis and development. We found significant advantages from using essential use cases in providing better traceability between requirements and responsibility-driven design[2].

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	

Figure 2: 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.)

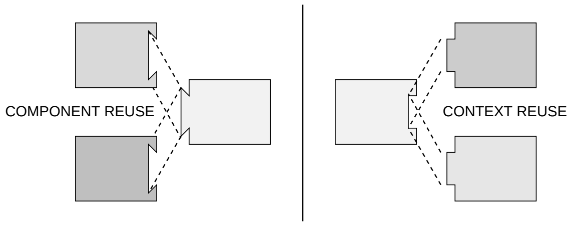


Figure 3: The invoking unit is the *context* and the invoked unit is the *component*. Component reuse involves several different contexts invoking the same component; Context reuse involves one context invoking different components using the same interface.

2.3 Modelling Reusability Support

In this section we briefly review a reusability support model for programming languages [3]. This model provides a set of structures and principles that explain how programming languages support code reusability.

The model addresses the kind of code arrangements where one unit of code *invokes* another unit of code. The invoking unit is called the *context* and the invoked unit the *component*. An important aspect of this structure is that support for reusability works both ways, as shown in figure 3. The common case is *component reuse*, shown on the left of figure, where two contexts both use a component. This reflects reusability of classes in a library. The other case is *context reuse*, shown on the right of the figure, where one context may invoke different components. This reflects reusability of frameworks [13].

When a context invokes a component, it has expectations as to what the component does. Similarly, the component has expectations as to how it will be invoked. These expectations are called *dependencies*, and they are the fundamental low-level concept of the model:

A dependency between two units of code is a condition that one unit meets so that it can use the other unit, or so that the other unit can use it.

Dependencies can adversely affect the reusability of a unit of code, because they limit the number of contexts that can invoke it, or the number of units it can invoke. The model identifies three techniques for addressing such dependencies: customisability, flexibility, and checkability.

flexibility: Flexibility is a property of a language that makes it possible to write code that is not specific to only one situation. For example, in object-oriented languages, polymorphism supports flexibility.

customisability: The customisability of a component is the amount of control a context has over how the context dependencies are met, and vice-versa. For example, parameters provide different ways for the context calling a component function to dictate how the function behaves to meet the context's requirements.

checkability: Where there are dependencies, it is desirable if it can be checked that the dependencies have been met. For example, in procedure calls the parameters represent dependencies, and checkability can be supported by the requirement that values passed match the parameters in number and type.

The model defines the reusability of a unit in terms of the amount of context or component reuse possible, affected by the dependencies of the unit, and strategies to address these dependencies.

3 Reusable Use Cases

In this section, we will discuss how the reusability support model above might apply to use cases. We will take each of the concepts in the model, and explore how they might be applied to make use cases more reusable. Whereas the model originally addressed the support for reusability in programming languages, the following discussion involves the "language" of use cases. We begin by discussing why essential use cases attracted our interest. In section 4 we will show how we applied all these ideas in providing tool support for reusable use cases.

3.1 Essential Use Cases and Reuse

The key characteristic of essential use cases (EUCs) is the abstraction of details about the technology involved in the interaction between user and system. The reason for this abstraction is to assist in capturing the requirements without making premature commitments to any particular details of the user interface. As we have become more familiar with EUCs, we have realised that they also facilitate reuse.

The abstract representation of interaction in EUCs is useful in analysis and when capturing requirements. In actual system development, however, the interaction will eventually be specified concretely as "enactments" of the EUCs. For example, the EUC on the right of figure 2 could lead to the concrete use case on the left. Concrete use cases can then be used to facilitate implementation. One set of EUCs can lead to several very different concrete enactments and implementations, depending on particular user interface requirements, such as software or hardware platforms. So, in this way, EUCs facilitate reuse because they allow many different concrete systems to be specified with one set of EUCs. One set can be used over and over to generate different concrete systems on different platforms. By the definition given in section 2.3, this means EUCs are more reusable than conventional use cases.

In fact, the abstraction sometimes means that some EUCs may apply not only to different concrete versions of the same system, but may also be useful in specifying completely different systems. Our experience is that there are some common kinds of use case that in the abstract form of an EUC do not mention any context of the system or domain at all. In particular, this is true for small units of simple interaction such as alarm notification, queries and confirmations, and so on. We have begun to identify these as *patterns* for EUCs [1], following the approach taken in documenting object-oriented design patterns [8].

3.2 Use Cases, Contexts, and Components

Recognising EUCs as offering the advantages of reusability, we decided to try and apply the reusability support model as a way of exploring ways to make use cases even more reusable.

Use cases themselves can be regarded as the units of reusability, and we can see that the main principles of context and component reuse also apply. When wishing to specify a new system, one might take a use case from elsewhere, and reuse it in new system; this is component reuse because it reuses the use case as a component. Alternatively, one might start with a system already specified, and create new use cases as well as or instead of the ones already there; this is context reuse because it reuses the existing context.

As well as building system requirement models based on use cases, some approaches involve use cases invoking other use cases. For example, in UML [9, 19], use cases can *include* or *extend* other use cases. These two invocation mechanisms allow articulation so that use cases can be used to describe single sequences of interaction, but do not require excessive redundancy or over-emphasis of exceptional behaviour. These invocation structures also allow reuse, because several different use cases may include the same use case, and a use case may also extend several other use cases.

3.3 Dependencies

The reusability of a use case is affected significantly by dependencies within it. A use case will be of limited reusability if it contains references unique to the system, or even the application domain, for which it was created. The reusability support model suggest several ways to address such limitations.

As with program code, a primitive but sometimes useful way to reuse units while addressing dependencies is cut-and-paste, or more correctly *copy-and-edit*. That is to say, one can simply copy the text of a use case, and then edit the copy, reducing dependencies or modifying them to suit the new circumstances. This is crude, but does at least permit reuse, and allows great latitude in recognising and addressing dependencies.

But the disadvantages of copy-and-edit are strong. Some problems involve the edit step. Dependencies are not typically easy to identify, and so careful understanding and assessment of all the text is required. Also, because the edit power is unrestricted, it is also possible to introduce problems where none previously existed.

Other problems with copy-and-edit involve the copy step. Some of the advantages of reuse concern the potential for sharing, rather than copying, the same unit. For example, much reuse of code is done by *calling* or *linking to* a code unit, rather than copying. If the unit is improved, by enhancement or debugging, all those sharing can easily benefit, and this in turn justifies increased investment in enhancement or debugging. This would be worthwhile when sharing use cases between systems or domains, but especially in the common case within a system where a use case may be reused by being included or extending other use cases within the same system. To share use cases, however, we need to address dependencies so that exactly the same the use case can be used in different situations.

3.4 Flexibility and Customisation

The reusability support model identified two strategies for addressing dependencies in a single unit: flexibility and customisation.

Flexibility is the ability to work with a general category of things, rather than only one specifically. In programming languages this includes polymorphism, where program code refers to some class or procedure, but exactly which class or procedure is not absolutely determined until necessary.

Flexibility is the principle by which EUCs facilitate reuse. Dependencies are what Constantine and Lockwood are referring in the passage quote earlier: "conventional use cases typically contain too many built-in assumptions, often hidden or implicit" [6]. The abstraction in the steps of an EUC works in a similar way to polymorphism. The abstraction means that dependencies between the use case text and the implementation technology are reduced, and this means that the EUCs are themselves more reusable and may be used in specifying different systems. At the same time, the abstract terms allow EUCs to apply to various later use cases in concrete interaction specification.

The reusability support model also identifies that dependencies can be reduced by allowing customisation. Customisation is the ability to change parts of a unit whenever it is invoked. In programming languages this includes parameter passing, where parameters are used as place-holders within a unit of code, and values for the parameters are passed when the unit is invoked.

One approach to customisation appropriate to use cases is simple textual parameterisation. This is the approach taken by text macro processors, which have been used for some time to support software reusability and especially portability [4]. These allow arbitrary text to be specified with embedded parameters, and later invocation with values specified for the parameters. The macro processor than generates a new unit of text, substituting the values for the parameters in the original text.

With a text macro processor, it would be possible to write use case text with embedded parameters for specific details, so eliminating dependencies. The use case could then be invoked for a particular system by specifying the values for the parameters, thus employing a reusable use case to yield a use case specific to the system.

3.5 Checkability

The reusability support model identifies dependencies as detrimental to reusability. We have discussed above how flexibility and customisability allow us to reduce dependencies; another strategy to address dependencies is checkability.

By checkability, we mean the ability to easily check whether dependencies are met when units are reused. In programming, we often reuse code that contains dependencies, whether global variables, or values passed to parameters. We can sometimes check that these dependencies are met by comparing names and types. Such checking is not strictly required, as can be shown with low level typeless programming languages, but checking does help reduce errors that are otherwise potentially difficult to detect and costly to correct.

If we are going to reuse use cases, there is similar potential for problems in accidentally combining use cases that do not work together. For example, if we allow customisation of use cases, as discussed above, there are problems that will arise should parameter values be inappropriately specified. We therefore suggest that, in any plan to reuse use cases, we should support checkability. Of course, the typical descriptions of use cases are not formal, and so checkability can not be provided with great rigour. The experience in programming languages indicates, however, that checkability can still be useful even if it simply uses visibility and redundancy to help avoid typographical errors and similar slips.

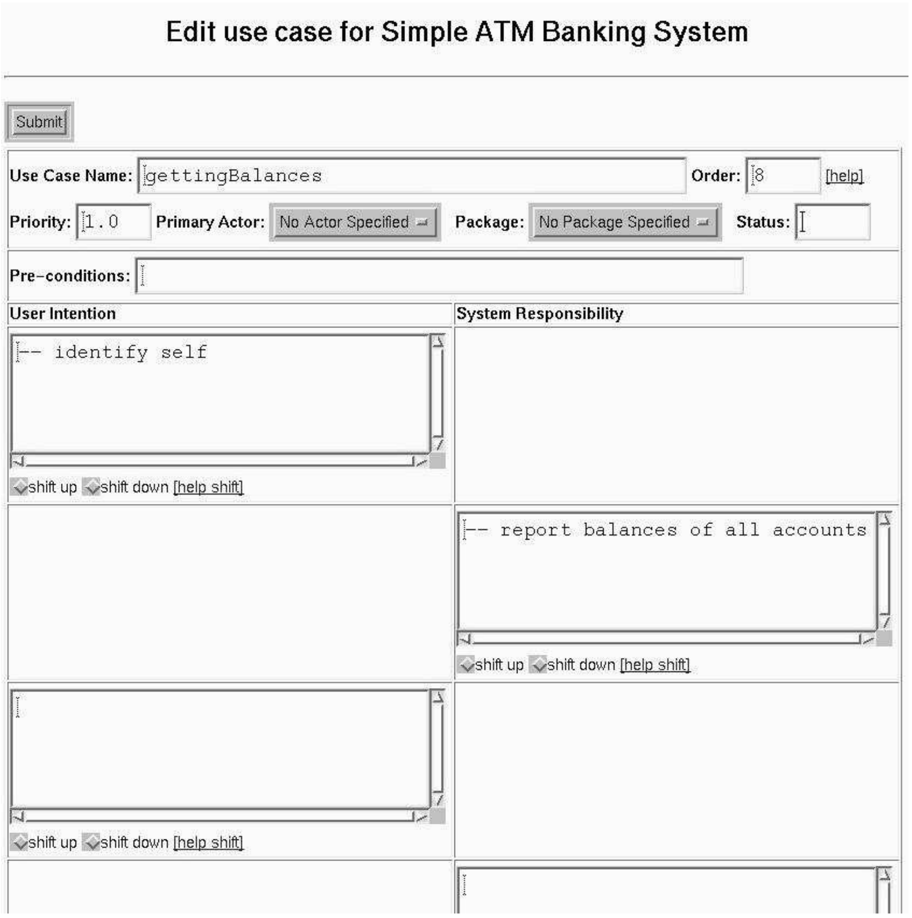


Figure 4: Entering the details of an essential use case in Ukase.

4 Ukase: Tool support for Reusable Use Cases

In the previous section, we gave a theoretical discussion on how use cases could be made more reusable. In this section, we show how the theory has been translated to practice, by presenting Ukase[1], a tool we have developed that supports reuse in essential use case models. The original purpose of Ukase was just to make the recording of use case models easier, but we quickly found it was useful for the development of the models themselves, and that in order to do development, better support for reuse was desirable. Below, we give an overview of what the tool does and describe the forms of reuse it currently supports.

4.1 Tool overview

Ukase is a *web-based* application. That is, one uses the tool by using a web browser to go to a URL. It is implemented as a Perl [21] CGI script [7], with the information stored in a MySQL database [15]. It is highly accessible (anywhere on the Internet), and may be used as group-ware (multiple people can access a model simultaneously). It is light-weight, that is, requires little effort to learn to use effectively, and imposes few restrictions on how it can be used.

As mentioned above, Ukase provides support for recording essential use cases. As such, it provides a mechanism for entering the details of the essential use cases, and a mechanism for

[1] *Ukase: Edict of Tsarist Russian government; any arbitrary order. (Concise Oxford Dictionary)*

Use Cases

1. **makingDeposit** (Customer) Priority: 2.0

Pre-conditions: ATM has deposit capability	
User intention	**System Responsibility**
-- specify identity and amount	
	-- confirm amount -- request deposit envelope
-- indicate envelope has been deposited	
	-- confirm deposit -- print message indicating when deposit will be available
Not all ATMs will accept deposits	

2. **gettingCash** (no actor specified)

User intention	**System Responsibility**
-- Identify self	
	-- verify identity -- offer choices
-- choose	
	-- despense cash
-- take cash	

3. **gettingBalances** (no actor specified) Priority: 1.0

User intention	**System Responsibility**
-- identify self	
	-- report balances of all accounts

4. **listingAccounts** (Customer)

User intention	**System Responsibility**
	-- List all Accounts in alphabetical order

5. **performingSelfTest** (Servicer)

User intention	**System Responsibility**
	-- report status of cashbox -- report status of keypad -- report status of network connection

Figure 5: Showing all the details of the use case model.

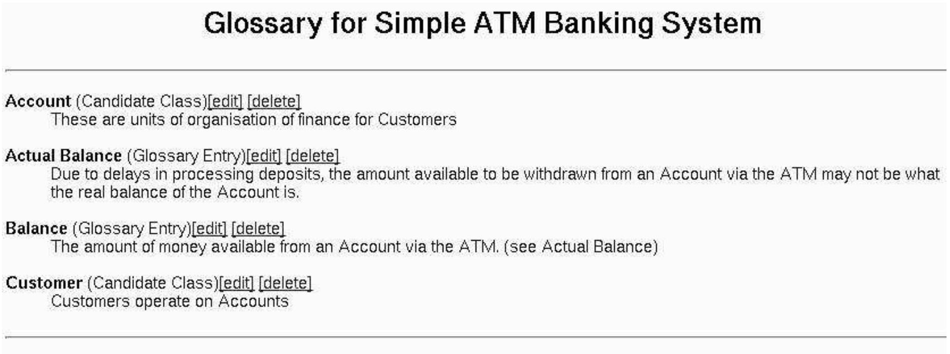
Simple ATM Banking System
Use Cases grouped by Actor

Actor Customer
1. [Edit] – listingAccounts
2. [Edit] – makingDeposit

Actor Servicer
1. [Edit] – performingSelfTest

No Actor specified
1. [Edit] – gettingBalances
2. [Edit] – gettingCash

Figure 6: Listing the use cases grouped by primary actor.


Glossary for Simple ATM Banking System

Account (Candidate Class)[edit] [delete]
These are units of organisation of finance for Customers

Actual Balance (Glossary Entry)[edit] [delete]
Due to delays in processing deposits, the amount available to be withdrawn from an Account via the ATM may not be what the real balance of the Account is.

Balance (Glossary Entry)[edit] [delete]
The amount of money available from an Account via the ATM. (see Actual Balance)

Customer (Candidate Class)[edit] [delete]
Customers operate on Accounts

Figure 7: The glossary, showing some of the terms that might be useful in the domain model.

displaying the results. The input is done using a CGI form laid out the way that a use case would look in a report (figure 4). As well as the steps of the interaction, users may specify pre and post conditions, and there is also an area for notes and comments.

As well as the name of the use case and its interactions, Ukase also allows users to organise the use cases (by package, or by primary actor) and some basic support for incremental development (priority). We are also experimenting with other meta information to help with organisation (status and order).

Ukase provides a variety of ways to organise use cases. Developers will typically use the view shown in figure 1, and once development is complete, the resulting model is available as a report (figure 5). Use cases can also be listed in a variety of ways (figure 6, for example).

Any use case model typically uses many terms particular to the application domain, which can make the intent difficult to understand. Ukase provides a *glossary*, where such terms can be more fully explained. Some of these terms are often concepts that would show up in an *analysis model*, and so Ukase allows such terms to be tagged as "candidate" classes (figure 7). While the glossary is useful in its own right, it also turns out to be useful for supporting reuse, as we show below.

The first version of Ukase was first in use about August 2000, and since then it has been used for a commercial case study, two research projects, and in three courses (final year undergraduate and graduate). It has also undergone heuristic evaluations [16] for usability, and the students in the graduate course critiqued its usefulness it as part of a course report. These evaluations confirm its usefulness, in that it makes the development and management of use case models much easier than doing so manually, and its usability, in that it most of its functionality is easy to learn and use.

4.2 Support for reuse

As discussed in section 3, we see a number of ways to support reuse. Ukase provides some of this support. One form of support it provides is to act as a *repository* of use cases. By providing access to all the use case models, it allows easy browsing, which can provide inspiration for use cases in models being created. It also helps in finding *patterns* in use cases [1].

Interestingly, the most immediately useful form of support for reuse has been the use case equivalent of copy-and-edit. Any use case can be *copied* to a per-user clipboard, and then *pasted* into any use case model. The semantics of this operation are similar to that of any text editor or word processor. In particular, no relationship between the original and the copy is maintained. The copy is treated exactly as if a new use case had been created and it just so happens that the text is identical to the original. The copy can then be edited independently of the original.

In fact, this is often exactly what is wanted. We often want to create use cases that are very similar to each other, for example, following particular conventions for describing details the bodies of the use case. Using the copy-and-edit operation makes this very convenient. However, as we've discussed earlier, sharing also has its advantages and we are working on providing such support.

Where the Ukase copy-and-edit differs from that of a text editor is that it operates on whole use cases, rather than on arbitrary pieces of text, as would be the case in any editor. This supports the fact that the unit of reuse is the use case. We believe that having the copy-and-edit operation recognise semantic boundaries in this way significantly improves support for reuse, because it maintains the integrity of what is being reused. We have also been exploring this idea in programming environments [22].

Ukase also provides support for customisation of use cases through *parameterised* use cases as discussed in section 3. Users may specify parameters anywhere in the name or body of a use case using the convention: `#{parameter name}`, where `parameter name` identifies the parameter. The specification of values to use for a particular parameter is done as part of the editing of a use case. Parameters cannot be replaced by arbitrary strings. To provide some checkability, Ukase requires that only entries from the glossary may be used as values for parameters (see figure 8).

While this support for parameterisation of use cases is somewhat primitive, as with the support for copy-and-edit, it is surprisingly effective.

5 Related Work

Reuse of requirements is not a new idea. Any artifact of the software development process is a candidate for reuse [14, 20]. What we have done is to examine how one might enable reuse of a particular form of requirements, namely use cases.

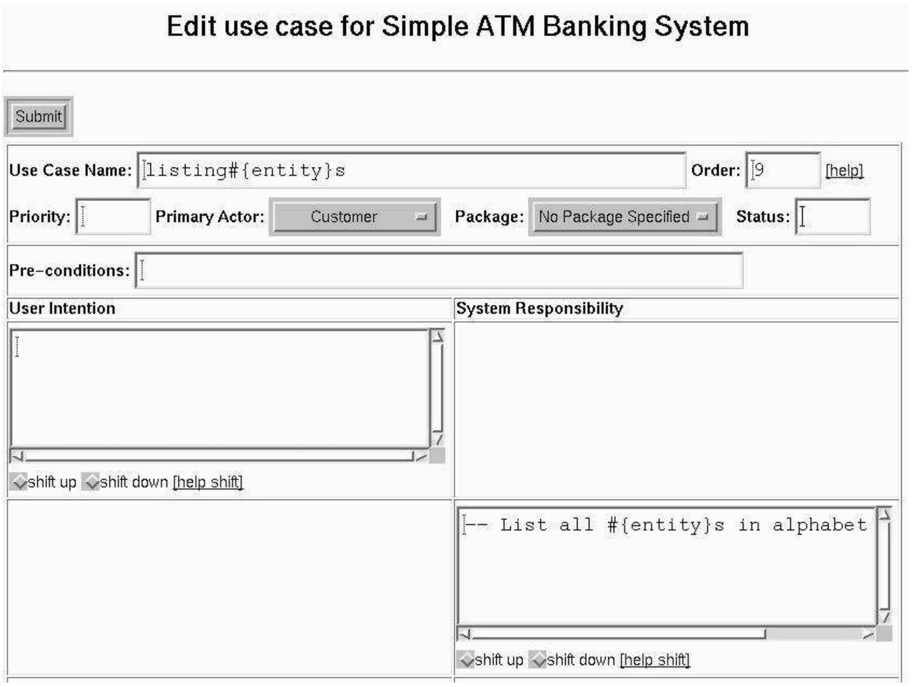
Use cases themselves were introduced as a key focus of Object-Oriented Software Engineering (OOSE) [11]. There are many forms for use cases, and we have been influenced especially by Wirfs-Brock discussing use cases as "conversations" [23, 24] The particular kind of use case we adopt in practice is the "essential use case", developed originally for analysis in the development of user interfaces [6]. Cockburn presents a survey of issues and approaches involving use cases [5].

The closest work to ours is by Jacobson et al. [12], who provide a useful framework in which to discuss and develop reuse processes. In their discussion, they identify *components* as workproducts engineered for reuse, so what we call reusable use cases they call use case components. Our development of reusable use cases parallels theirs, which serves to validate our model.

There are some differences. Jacobson et al. divide components into *concrete* and *abstract* components. Concrete components are ones that can be reused as is, whereas abstract components are incomplete in some way and must be specialised before they can be reused. Essential use cases do not fit into either of these categories. Early in the requirements capture process, essential use cases may capture exactly what is needed, and so be used as is. However, later in the process they get refined into concrete enactments, and so the originals may also be regarded as "incomplete".

Jacobson et al. also discuss the use of parameters in use cases, however, their use of parameters is to provide generalisation, whereas our use provides customisation. In fact, the mechanisms are very similar, and can be used for both purposes. We also address the need for checkability when parameters values are specified.

(a)



(b)

(c)

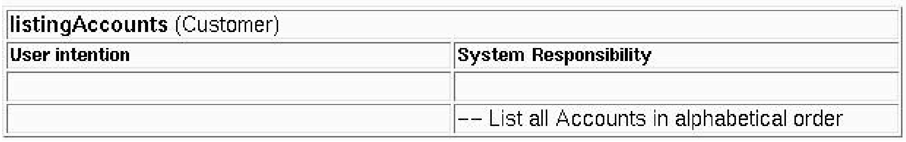


Figure 8: Part (a) shows how the user names the parameter `entity`. Whenever the set of parameters for a use case changes, the user is required to assign values to the parameters when the changed use case is submitted (b). Part (c) shows the resulting use case, with the glossary item `Account` substituted for `entity`.

The best tool support for management of use cases that we are aware of is Requisite Pro [18]. This tool manages conventional use cases as documents, and provides traceability between other related documents (such as a Vision document), and to object-oriented development models supported by Rational Rose [17]. While Requisite Pro does act as a repository for use case models, it provides no other direct support for reuse. Our tool also takes a more light-weight approach, and so is useful for the early stages of system analysis.

6 Conclusions

We have discussed what reusable use cases might mean, how use cases may be made more reusable, and what tool support would be appropriate for reusing use cases. We used the reusability support model to guide our discussion. Our approach has been quite pragmatic; we want to reuse use cases in the same way we might reuse code. We presented Ukase, a tool that supports the reuse of use cases.

References

[1] Robert Biddle, James Noble, and Ewan Tempero. Patterns for essential use cases. In *Proceedings of KoalaPLoP 2001*, 2001.

[2] Robert Biddle, James Noble, and Ewan Tempero. Essential use cases and responsibility in object-oriented development. In *Proceedings of the Australisian Computer Science Conference*, January 2002.

[3] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*, pages 52–61. IEEE Computer Society, April 1996.

[4] P. J. Brown. *Macro processors and techniques for portable software.* Wiley, January 1974.

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

[6] 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.

[7] National Center for Supercomputing Applications (NCSA). The common gateway interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[8] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[9] Object Management Group. Unified modeling language (UML) 1.3 specification, 2000.

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

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

[12] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997.

[13] Ralph E Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[14] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[15] Mysql. http://www.mysql.com.

[16] Jakob Nielsen. *Usability Engineering*. Academic Press, New York, 1992.

[17] Rational Software. Rational Rose. http://www.rational.com/products/rose/.

[18] Rational Software. Requisite Pro. http://www.rational.com/products/reqpro/.

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[20] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, 1995.

[21] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[22] Glen Wallace, Robert Biddle, and Ewan Tempero. Smarter cut-and-paste for programming text editors. In *The Austrialasian User Interface Conference*, pages 56–63, Gold Coast, Australia, January 2001. IEEE Computer Society.

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

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