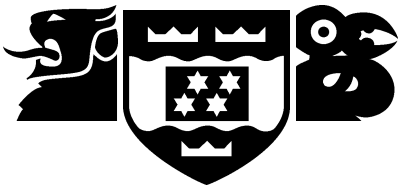
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

From Essential Use Cases to Objects

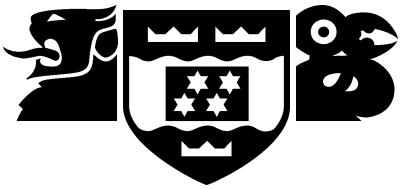
Robert Biddle, James Noble, and Ewan Tempero

Technical Report CS-TR-02/12
July 2002

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

From Essential Use Cases to Objects

Robert Biddle, James Noble, and Ewan Tempero

Technical Report CS-TR-02/12
July 2002

Abstract

One of the main motivations for essential use cases was the context of user interface design. We, however, have been exploring the application of essential use cases in general object-oriented system development. Our experience has been very positive, and we found advantages to essential use cases that assist in both analysis and in design. This paper outlines two techniques involving essential use cases: use of role-play in requirements analysis, and distribution of system requirements from essential use cases to objects.

Publishing Information

A later version of this paper is to be presented at forUSE 2002, The First International Conference on Usage-Centered Design, Performance-Centered Design, and Task-Oriented Design - 25-28 August 2002, Portsmouth, New Hampshire, USA. This paper brings together an overview of material covered in more detail in other publications; this is all described within the paper.

Author Information

Robert Biddle and James Noble are members of the academic staff in the School of Mathematical and Computing Sciences and the School of Information Management at Victoria University of Wellington. Ewan Tempero is a member of the academic staff of the Department of Computer Science at the University of Auckland.

From Essential Use Cases to Objects

Robert Biddle[a]
James Noble[a]
Ewan Tempero[b]

[a]Victoria University of Wellington, New Zealand
Email: robert,kjx@mcs.vuw.ac.nz

[b]University of Auckland, New Zealand
Email: ewan@cs.auckland.ac.nz

Abstract

One of the main motivations for essential use cases was the context of user interface design. We, however, have been exploring the application of essential use cases in general object-oriented system development. Our experience has been very positive, and we found advantages to essential use cases that assist in both analysis and in design. This paper outlines two techniques involving essential use cases: use of role-play in requirements analysis, and distribution of system requirements from essential use cases to objects.

1 Introduction

Identification and employment of use cases is now common practice in software development, and the use case is now a recognised concept in modeling languages and in development processes. Essential use cases were introduced as a refinement of the general concept, originally proposed to address needs for technology independence in user interface design. We feel that essential use cases deserve a wider role, and have been exploring the application of essential use cases in general object-oriented software development.

Our motivation to consider essential use cases for general system development began with some simple advantages. In particular, their abstraction allows early consideration of interface details to be avoided, and also allows brevity. As we applied essential use cases in developing a variety of systems, however, we came to realise there were a number of deeper benefits.

We began using essential use cases as our prime requirements gathering tool, and have now used this approach for a number of system development projects over the last three years. We claim that essential use cases are suitable for object-oriented software development in general, and indeed have significant advantages over conventional use cases. We present a detailed discussion of the advantages elsewhere (Biddle, Noble and Tempero 2002a).

In this paper, we briefly review the nature of essential use cases, and then discuss two of the new techniques we have been exploring. The first technique concerns requirements analysis, and involves the use of roleplay for requirements review, which is supported by the nature of essential use cases. The second technique concerns design, and involves the distribution of system responsibilities identified in the essential use cases, yielding a set of objects and their responsibilities constituting an object-oriented design traceable to the requirements.

2 Background: Essential Use Cases

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. The definition used in the Rational Unified Process is that 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” (Jacobson, Booch and Rumbaugh 1999). Identification of use cases is beneficial throughout the system development, beginning with early analysis and continuing thorough to testing and maintenance.

gettingCash

USER ACTION	SYSTEM RESPONSE
insert card	
	read magnetic stripe request PIN
enter PIN	
	verify PIN display transaction option menu
press key	
	display account menu
press key	
	prompt for amount
enter amount	
	display amount
press key	
	return card
take card	
	dispense cash
take cash	

Figure 1: A conventional use case for getting cash from an automatic teller system. (From Constantine and Lockwood.)

Essential use cases are part of Usage-Centered Design, as developed by Larry Constantine and Lucy Lockwood (Constantine and Lockwood 1999). Constantine and Lockwood support use cases, and agree with many claims about their advantages. They also see limitations: "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." This is problematic for UI design both because it forces design decisions to be made very early, and because it then embeds these decisions in requirements, making them difficult to modify or adapt at a later time.

Essential use cases were designed to overcome these problems. The term "essential" refers to essential models that "are intended to capture the essence of problems through technology-free, idealized, and abstract descriptions". Constantine and Lockwood define an essential use case as follows:

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

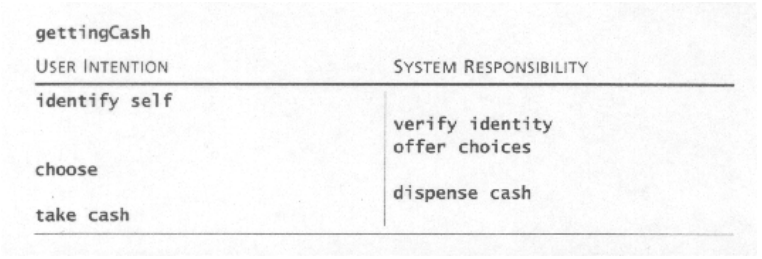
Essential use cases are documented in a format representing a dialogue between the user and the system. This resembles a two-column format used by Wirfs-Brock (Wirfs-Brock 1993). In Wirfs-Brock's format, the column labels refer to the *action* and the *response*. Although Wirfs-Brock does discuss a framework for levels of abstraction in use cases, the two-column use cases she presents do comprise concrete steps of interaction between a user and a system.

In contrast, the essential use case format labels the columns *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 figures 1 and 2. The dialogue in figure 1 is for a conventional use case, described in terms of actions and responses. The dialogue in figure 2 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.

3 Essential Use Cases and Requirements

Our first exploration in using essential use cases was in working with system requirements. We wanted to make this activity more accessible to newcomers and non-technical stakeholders, and we wanted the activity to be done in a



gettingCash

USER INTENTION	SYSTEM RESPONSIBILITY
identify self	
	verify identity
	offer choices
choose	
	dispense cash
take cash	

Figure 2: An essential use case for getting cash from an automatic teller system. (From Constantine and Lockwood.)

way where a variety of people worked together as a team.

For some time now, we have used and admired the CRC (class-responsibility-collaborator) card technique (Beck and Cunningham 1989) for the later stages of object-oriented development. To make use cases more accessible, we decided to take a similar approach. Our basic idea was to use index cards for use cases, and to somehow involve roleplay as well. This section presents an overview of the ideas: more detailed discussion is presented elsewhere (Biddle, Noble and Tempero 2001).

3.1 Use Case Cards and Roleplay

We decided that each use case card should represent a single use case, should show the name of the use case, and the steps of the dialogue script. We follow the canonical presentation of an essential use case, and to distinguish the two roles, we split the card down the centre, and write the user's lines on the left, and the system's lines on the right. We initially drew a straight vertical line down the centre of the card, but found over time this made it difficult to distinguish use case cards from CRC cards. After some experimentation, we decided to draw a jagged line down the centre, which suggests interaction, and makes it easy to tell use case cards apart from CRC cards. (See figure 3.)

The essential use case style offered us several advantages. Firstly, the two-column format facilitates roleplay because a use case in this easy dialogue form resembles a simple script. The script has two roles, user and system, so use case roleplay can simply involve two people, each with a part in the script. Another important stylistic advantage concerns the abstraction in essential use cases. The abstraction keeps essential use cases brief, and so able to fit on an index card. Also, the abstraction helps avoid unnecessary debate about irrelevant implement details, so allowing more rapid progress to be made. The focus on intention and responsibility also has important consequences that we discuss later.

Use case cards and roleplay assist primarily in elaborating use case "bodies", determining the steps of the interaction. Before that can happen it is first necessary to identify the use cases. This initially involves background analysis to come up with suggestions for the different ways users may interact with the system. In large system development, even use case identification can constitute an activity of significant size and scope. We use analysis techniques to identify a wide variety of candidate use cases, and then prioritise them, selecting a few "focal" use cases that represent critical interactions that will be necessary even in early prototypes.

Our roleplay techniques then begins. We start to work on these focal use cases by elaborating their steps with cards and roleplay. We generally suggest doing this with a small team of 3 to 6 people, similar to that recommended for CRC cards. We begin to work with cards by writing brief names for the focal use cases on the cards. For example, a banking system might have focal use cases called depositing cash, checking account balance, and withdrawing cash.

We then select a card and begin to work out the dialogue. We select people for the roles of user and system, and use an exploratory kind of roleplay rehearsal. The team works together on determining the steps in the dialogue. When ideas seem reasonable, the roleplayers "play-act" through the script. The rest of the team audit, and afterwards the dialogue is discussed and improved where necessary. Increasing availability of document cameras means cards can easily be projected on a large screen for larger groups to follow. (See figure 4.)

This process is applied iteratively until the team is satisfied that the dialogue represents the way the user and the system should interact. Sometimes it helps to leave one use case and work with another, returning later to improve the earlier one.

One use of roleplay is for exploratory and iterative development of the use cases. We also use roleplay as a way

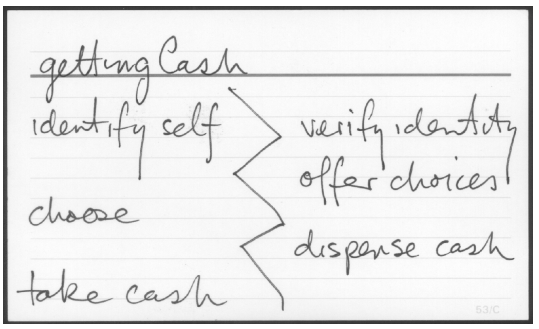


Figure 3: An Essential Use Case Card

Figure 4: Performing a Use Case Roleplay with Use Case Card Shown via Document Camera

of presenting use cases for review. Such reviews may be conducted by peer developers, by people working on other aspects of a project, or by stakeholders or experts in the system domain.

After the index cards and roleplay have helped determine the use case bodies, the full details may be recorded in requirements documentation or in Case tool databases. The use cases can be used to drive system design and review, and again later in system testing and demonstration. The cards and roleplay may still prove useful in some later activities, where their immediate and lightweight nature support rapid review and exploration of interaction alternatives.

3.2 Experience and Discussion

Our experience with essential use cases has shown that the simple benefits are realised, and there are also more profound effects. As we had hoped, the abstraction does keep the use cases brief enough for an index card, and also helps avoid premature debate about implementation.

We had explicitly sought active and immediate engagement by team members, and that has happened as expected. As with CRC cards, the concrete element of the index cards, together with the behavioural element of the roleplay, result in a focus of attention and command active participation. This alone is valuable because it ensures a focus on determining requirements that involves the whole team.

The index cards also help in a way that, as with CRC cards, relates to their simple nature as cards: they are discrete and concrete, and only a lightweight investment. Cards can be arranged or prioritised on a table top, needed cards can be selected while leaving others, and cards can held during roleplay. A card can be changed or discarded easily without disturbing others, and the change can made immediately.

The advantages of roleplay go deeper. People, even non-developers, are good at following dialogue. Moreover, they are familiar with the dramatic device of roleplay and the willing suspension of disbelief on which drama depends. People are typically able to watch roleplay, and imagine the interaction with the finished system with real understanding. This is a form of fast prototyping, and it allows fast review and feedback that allows iterative improvement.

On a deeper level still, we have seen there is heuristic merit in casting the user role as expressing "intention", and the system role as expressing "responsibility". Together, these lead to use case dialogues that better reflect the real motivation of the user, and better capture the requirements of the system, while at the same time avoiding premature design decisions.

In our experience, one especially critical early issue in determining requirements is simply determining the boundary of the system, distinguishing what the system should do from what it should not do. It can be very difficult to reach the agreement necessary to make such distinctions while involving all the people involved, analysts and stakeholders.

Use case cards and roleplay have proven very useful in determining the system boundary. Our approach is to apply an approach familiar in design: we regard the system as a "black-box". The internal workings are not specified, but the way the system is used is specified by the use cases. The two-column dialog form of each use case card clearly shows the role of the user distinct from the role of the system, and the line dividing these roles is the boundary of the system.

There are several ways in which use case roleplay assists to determine the system boundary. In early exploratory roleplay, it can quickly become clear if team members differ in their understanding about what the system is required to do. People reading the same analysis documents can come up with different interpretations, and it is especially common for differences to arise between technical designers and business analysts or domain experts. It is important to detect and resolve such differences early, and determining actual interaction sequences for roleplay tends to expose these differences.

The exploratory roleplay can also expose unreasonable assumptions about both the user and the system roles. For the user, it can become apparent that actions specified in a use case are inconsistent with how an actual user is likely to behave. For the system, it can become apparent the required behaviour is not possible because critical information will not be available. Actually play-acting the roles seems to make such anomalies more obvious than if the use case was just read carefully by someone alone.

To resolve such anomalies, the use case steps may have to be modified, or the use case may need to be structured in a different way. Such changes may require modifications be made to other use cases, and may even require even creation of new use cases. The important result is that problems can be identified and fixed at this early point, rather than at later more expensive points in development.

Roleplay also makes use case interaction understandable to observers outside a development team. This is especially useful in presenting use cases to busy stakeholders. Roleplay is quick, immersive, and very accessible. Without heavy investment, it makes it possible to discuss issues, compare alternatives, and possibly detect flaws.

We have found there are some considerations needed in working with essential use cases. One is that use case roleplay will itself be abstract, and so not as realistic as if it involved specific interface details. In our experience this has not been problematic, although teams do find it helpful to sometimes consider concrete enactments of the use case to check their understanding. The origin of essential use cases is in user interface design. In typical practice, the system responsibilities are limited to responsibilities directly concerning interaction. The semantics of deeper responsibilities for system behaviour are typically left unexplored. We have found, however, that it is possible to include these deeper responsibilities where helpful. For example, the use case in the earlier figure might well involve responsibilities to maintain account balance integrity and to audit trail entries.

We have also been exploring tool support for requirements analysis with essential use cases. In particular, we have developed a web-based tool that facilities recording essential use cases, and also assists in using essential use cases to begin the design process (Biddle, Noble and Tempero 2002c).

4 Essential Use Cases and Design

One of the distinguishing features of essential use cases is the abstraction in the dialog steps, where the user is associated with "intention" and the system is associated with "responsibility". In object-oriented design, the term "responsibility" already holds a special role. In particular, responsibility is the pivotal concept in CRC cards, and in Responsibility-Driven Design. The identification of responsibilities in essential use cases therefore affords a new way to link use cases to object-oriented designs.

In the CRC card technique, class responsibility helps determine system articulation, and this originally pedagogical tool is now seen as useful in the context of overall system development (Wilkinson 1996, Bellin and Suchman Simone 1997). In responsibility-driven design (Wirfs-Brock and Wilkerson 1989, Wirfs-Brock, Wilkerson and

Wiener 1990) the idea of responsibility is used more thoroughly and on a larger scale. Responsibilities are associated with objects, and represent knowledge an object maintains, or actions an object can perform. Responsibilities thus emphasize abstract behavior while being silent about possible implementation structure.

Both CRC and responsibility-driven design involve the concept of "responsibility" to guide making design decisions. In particular, these techniques suggest that in an object-oriented system, each object or class should have a coherent and well-understood set of responsibilities. In this way, responsibility is used as a heuristic to partition a system into collaborating objects with the overall responsibilities distributed reasonably.

A basic principle of object-oriented design is that objects involve behaviour and information that work together. Responsibility is a good heuristic for determining this, because the word "responsibility" usually suggests both a duty to do something, and the resources with which to do it. Responsibility also allows delegation, allowing large responsibilities to be managed by delegating smaller responsibilities to others.

Responsibility involves both abstraction and encapsulation, as Wirfs-Brock et al.(Wirfs-Brock and Wilkerson 1989) explain:

> *The responsibility-driven approach emphasizes the encapsulation of both the structure and behavior of objects. By focusing on the contractual responsibilities of a class, the designer is able to postpone implementation considerations until the implementation phase.*

In essential use cases, the idea of a responsibility is to identify what the system must do to support the use case, without making commitments as to how it will actually be done. This resembles object encapsulation, where the internals of an object cannot be directly accessed from outside, and it has similar benefits. This role of responsibility in use cases is entirely consistent with the role of responsibility in design. Both describe behavior without describing implementation. This commonality presents valuable opportunities to link the way we work when determining requirements and the way we work when determining design.

4.1 Distribution of System Responsibilities

So responsibility is a powerful concept, and already familiar as a heuristic in object-oriented design. Essential use cases also involve responsibility as a similar heuristic. There seems to be a valuable opportunity to harness this connection. We can regard essential use cases as indicating a starting state for design. From this viewpoint, the "system" is like a single object. This a helpful attitude to foster while working out use cases, because it discourages concern for internal system design at too early a stage. Note that this is the same principle we used in requirements analysis, where we regarded the system as a "black-box" while determining the system boundary.

The idea of the system as a single object can be used as a way to begin design and then to develop design. We take the system responsibilities as documented in the essential use cases, and consolidate these as a coherent set. We then regard the system overall as an object having this set of responsibilities, in the same way that in CRC design each object has a set of responsibilities. The design process is then to distribute the responsibilities in this single object among a set of objects. We call this approach "distribution of system responsibilities" (DSR).

A typical CRC procedure would begin with a set of candidate classes or objects, each with an initial suggestion of responsibilities. The procedure would then follow use cases to explore the distribution, amending objects and responsibilities iteratively. The alternative DSR procedure would be to begin with the single system object and its responsibilities, and then begin to follow CRC. As with typical CRC, we would also identify a set of candidate objects and their responsibilities, and we would also explore the design as driven by use cases.

However, we begin with a working design consisting of a single object. We then explore by sharing the responsibilities between candidate objects, or delegating some responsibilities to candidate objects. As with more typical CRC, we explore and amend objects and responsibilities iteratively. This approach seems true to the spirit of CRC in every way, and merely provides a more definite starting state.

The advantage of DSR at the beginning is that we start out with the responsibilities from the use cases in mind, so that our initial CRC object roleplay addresses the system interaction from the essential use cases. As we progress, and especially as we near a complete design, the advantage is that the set of responsibilities of the objects must together still meet the responsibilities identified in the essential use cases.

In general, the advantage of DSR is traceability: we should we able to link the responsibilities of the design to the responsibilities identified in the essential use cases. There is also a related secondary advantage. While traceability allows us to check requirements against design, DSR also offers operational guidance while doing design. Thus at any point in the iterative design process, one possible approach to is to distribute the essential use case responsibilities further, or to redistribute responsibilities in a different way.

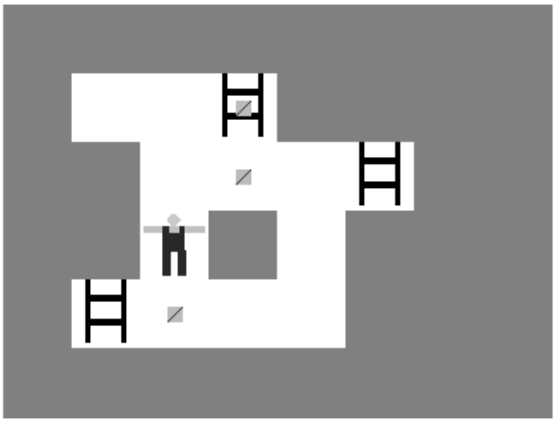


Figure 5: An example of what a Sokoban level might look like. Shown is the worker, three shelves, and three boxes. One of the boxes is on a shelf.

Of course, we acknowledge that strictly distributing system responsibilities is not a complete design technique. In particular, it does not have all the advantages of CRC cards in working from known to unknown objects, nor the advantages of responsibility-driven design in identifying abstract higher levels of the design. However, we do believe that DSR does offer useful advantages of both traceability and operational guidance that can work together with CRC cards or responsibility-driven design.

4.2 Example: Sokoban

In this section, we present a small detailed example of working from a set of essential use cases toward a workable object-oriented design via distribution of system responsibilities. The example is a program to play the game "Sokoban". In this paper we just present this example to support the discussion above. More details on this particular design exercise are available elsewhere (Biddle, Noble and Tempero 2002b).

Sokoban is a type of puzzle involving "boxes", "shelves", and "walls" in a warehouse. The object is to place all boxes on shelves when the only operation available is "push box". There are several variations. The one used for this case study is as follows:

> The Sokoban game consists of a set of levels. Each level describes a warehouse, consisting of a set of walls, shelves, and boxes, and the starting position of the worker. The worker may move into any empty location, or may move into a location occupied by a box, provided the location beside the box opposite from the worker is either empty or contains an empty shelf. That is, workers may only "push" boxes. Once all the boxes are on shelves, the level is completed, and the next level is started. Once all levels are completed the game is over. If this happens without the player either restarting a level, or skipping any levels, then they may enter their name in the Hall of Fame.

This is the kind of description that might be given to a student in a first year programming course, and, as a consequence, it is perhaps more carefully worded than might normally be the case. Figure 5 shows what a solution may look like.

This section describes the requirements for the Sokoban system as a set of essential use cases (EUCs). The convention with EUCs is to be as concise as possible in describing the bodies of the use cases. Thus, they typically do not begin with an action such as "choosingOperation", as that is implied by the EUC name. Also, it would usually be the case that the *actors* related to the use cases would be specified, but in this case there is only one actor (the player). The essential use cases for Sokoban are shown below.

StartingGame	
USER INTENTION	SYSTEM RESPONSIBILITY
	• Request the player's name
• Provide name	
	• Record player's name • Set the current Level to be the first Level • Set the current Level to its initial configuration • Show the current Level

RestartingLevel	
USER INTENTION	SYSTEM RESPONSIBILITY
	• Set the current Level to its initial configuration • Show the current Level • If the current level is any level other than the first one, record that the player cannot enter the Hall of Fame

OpeningLevel	
PRE-CONDITIONS: Level specification must be valid.	
USER INTENTION	SYSTEM RESPONSIBILITY
• Specify level	
	• Read the description of the specified Level • Set the current Level to be the specified Level • Set the current Level to its initial configuration • Show the current Level • If the current level is any level other than the first one, record that the player cannot enter the Hall of Fame

MovingWorker	
PRE-CONDITIONS: Specified direction must be valid	
USER INTENTION	SYSTEM RESPONSIBILITY
• Specify direction	
	• If the Location adjacent to the Worker in the direction specified is empty, or it contains a Box and the Location adjacent to it in the specified direction is either empty or contains an empty shelf, then move the Worker in the direction specified, and if there is a box, move it as well • Show the current Level • If all the boxes in the Level are on a shelf, then load the next Level • If the current Level is the last one, and the player is eligible for the Hall of Fame, then record the player's name in the Hall of Fame

ShowingHallofFame	
USER INTENTION	SYSTEM RESPONSIBILITY
	• List all players in the Hall of Fame

We now present a series of iterations an object-oriented design for the Sokoban system. The designs are presented as a set of CRC "cards". The responsibilities in each class are labelled so that traceability between the requirements and the design can be tracked. This can be developed further, but for the moment we just note that whenever responsibilities are changed or delegated, we indicate their relationship to other responsibilities using these labels.

In the interests of space, we use the convention here that any responsibilities completely delegated to another class will not be shown in the delegating class. Responsibilities that are only partially delegated, will however still be listed (possibly changed in some way).

4.2.1 Iteration 1: The System Object

The first iteration of the design is the "system object", a single class whose responsibilities are those listed as system responsibilities in the EUCs, which means this design is guaranteed to provide the behaviour specified in the requirements. Here, we just list the responsibilities (in alphabetical order), as there are obviously no collaborators for this class.

System Object	
SO1	If all the boxes in the Level are on a shelf, then load the next Level
SO2	If the Location adjacent to the Worker in the direction specified is empty, or it contains a Box and the Location adjacent to it in the specified direction is either empty or contains an empty shelf, then move the Worker in the direction specified, and if there is a box, move it as well
SO3	If the current Level is the last one, and the player is eligible for the Hall of Fame, then record the player's name in the Hall of Fame
SO4	If the current level is any level other than the first one, record that the player cannot enter the Hall of Fame
SO5	List all players in the Hall of Fame
SO6	Read the description of the specified Level
SO7	Record player's name
SO8	Request the player's name
SO9	Set the current Level to be the first Level
SO10	Set the current Level to be the specified Level
SO11	Set the current Level to its initial configuration
SO12	Show the current Level

4.2.2 Iteration 2

Examining the System Object in section 4.2.1, we immediately note the frequent use of "Level". This suggests that providing a **Level** class may reduce the amount of work that the System Object is responsible for. Once we have done that, we determine how we might delegate some of the System Object responsibilities to the new class.

In doing this, we will be changing the responsibilities of the System Object. We change the name of the resulting class to better reflect its role in the design, namely the main interface to the application, to **Sokoban**. Because this class presents the same view of the system as the System Object, it should have the same set of responsibilities; it just may delegate some of them to other classes.

Sokoban	
SO1	If all the boxes in the Level are on a shelf, then load the next Level (partially delegated to LEV1)
SO2.1	Given a direction, move the Worker in that direction (replaces SO2, partially delegated to LEV4)
SO3	If the current Level is the last one, and the player is eligible for the Hall of Fame, then record the player's name in the Hall of Fame
SO4	If the current level is any level other than the first one, record that the player cannot enter the Hall of Fame
SO5	List all players in the Hall of Fame
SO7	Record player's name
SO8	Request the player's name
SO9	Set the current Level to be the first Level
SO10	Set the current Level to be the specified Level
SO11	Set the current Level to its initial configuration (partially delegated to LEV2)
SO12	Show the current Level (partially delegated to LEV3)
Collaborators: Level	

Level	
LEV1	Report whether all the boxes in the Level on a shelf
LEV2	Set to initial configuration
LEV3	Show all the details of the Level
LEV4	Given a direction, if the Location adjacent to the Worker in that direction is empty, or it contains a Box and the Location adjacent to it in that direction is either empty or contains an empty shelf, then move the Worker in the direction specified, and if there is a box, move it as well
LEV5	Read description and create

The decision that deserves some discussion is how SO2 gets distributed to **Level**. There are possibly a number of ways this distribution could have been done. Because this case study is not trying to produce the "best" design, but merely a "good" design, the focus here is on whether the decisions were reasonable, rather than the best.

In this case, we intend the **Level** class in this design to deal with all aspects of what has happened and can happen in a particular level. This includes tracking where the Worker is, and where Locations are with respect to each other. This being the case, it seems reasonable to delegate large parts of SO2 to this class.

4.2.3 Iteration 3

This iteration focuses on the **Level** class. Its repeated use of the term "location" suggests that a **Location** class would be useful. This has the following impact on the design (**Sokoban** is unchanged).

The design decision is that a **Level** will consist of a set of **Locations**. Each **Location** has a position (where it will be drawn) and contents. It is up to the **Level** class to keep track of how different **Locations** relate to each other.

Location	
LOC0	Given position and contents, be created
LOC1	Show details (render representation according to contents) at position
LOC2	Indicate contents (empty, Box, empty Shelf, full Shelf, Worker, Wall)
LOC3	Move Worker in or out (as specified)
LOC4	Move Box in or out (as specified); if the contents is an empty or full Shelf, then change as appropriate

Level	
LEV1	Report whether all the boxes in the Level on a shelf (partially delegated to LOC2)
LEV2.1	Read description of current level and from that create locations and assign positions and contents (replaces LEV2, partially delegated to LOC0)
LEV3.1	Have each location show its details (replaces LEV3, partially delegated to LOC1)
LEV4	Given a direction, if the Location adjacent to the Worker in that direction is empty, or it contains a Box and the Location adjacent to it in that direction is either empty or contains an empty shelf, then move the Worker in the direction specified, and if there is a box, move it as well (partially delegated to LOC2, LOC3, and LOC4)
LEV5.1	Given level identification, read description of specified level and from that create locations and assign positions and contents (replaces LEV5, partially delegated to LOC0)
Collaborators: Location	

4.2.4 Iteration 4

Returning to the **Sokoban** class, we note the need for maintaining details about the player, suggesting a **Player** class.

Player	
P1	Request and record name, and make eligible for the Hall of Fame
P2	Make ineligible for Hall of Fame
P3	Report name
P4	Report if eligible for Hall of Fame

Sokoban	
SO1	If all the boxes in the Level are on a shelf, then load the next Level (partially delegated to LEV1)
SO2.1	Given a direction, move the Worker in that direction. (partially delegated to LEV4)
SO3	If the current Level is the last one, and the player is eligible for the Hall of Fame, then record the player's name in the Hall of Fame (partially delegated to P3 and P4)
SO4	If the current level is any level other than the first one, record that the player cannot enter the Hall of Fame (partially delegated to P2)
SO5	List all players in the Hall of Fame
SO9	Set the current Level to be the first Level
SO10	Set the current Level to be the specified Level
SO11	Set the current Level to its initial configuration (partially delegated to LEV2)
SO12	Show the current Level (partially delegated to LEV3)
Collaborators: Level, Player	

4.2.5 Iteration 5

Finally, we create a **HallOfFame** class, which takes over more of the responsibilities of the **Sokoban** class. This is the final design in our case study. While other iterations may be possible, what we have is quite a good start — it is a design that we are quite confident satisfies the requirements and the responsibilities seem fairly well distributed. We could use this as a starting point for application of other techniques, such as responsibility-driven design.

HallOfFame	
HOF1	Read names of players (from SO3,SO5)
HOF2	Add name of player (from SO3)
HOF3	Write names of players (from SO3)
HOF4	Display names of players (from (SO5)

Sokoban	
S1	If all the boxes in the Level are on a shelf, then load the next Level (partially delegated to LEV1)
SO2.1	Given a direction, move the Worker in that direction. (replaces SO2, partially delegated to LEV4)
SO3	If the current Level is the last one, and the player is eligible for the Hall of Fame, then record the player's name in the Hall of Fame (partially delegated to P3, P4, HOF1, HOF2, and HOF3)
SO4	If the current level is any level other than the first one, record that the player cannot enter the Hall of Fame (partially delegated to P2)
SO5	List all players in the Hall of Fame (partially delegated to HOF1 and HOF4)
SO9	Set the current Level to be the first Level
SO10	Set the current Level to be the specified Level
SO11	Set the current Level to its initial configuration (partially delegated to LEV2)
SO12	Show the current Level (partially delegated to LEV3)
Collaborators: Level, Player, HallOfFame	

5 Conclusions

Use cases are seen as beneficial in many aspects of system development, and the refinement of essential use cases was originally made to address needs in user interface design. We have been exploring the application of essential use cases in general system design.

Our initial motivation concerned straightforward aspects of essential use cases. Essential use cases support roleplay well, because of the use of dialogue. They are brief, and can be developed quickly, because they avoid unnecessary debate about implementation details. We also found deeper advantages that relate to many aspects of system development. Most importantly, the abstraction of essential use cases comes from identifying user intentions and system responsibilities, and these both act as heuristics that benefit software development.

Some of the advantages flow from the heritage that essential use cases have from user interface design. In requirements analysis, therefore, they emphasise the dialogue of interaction, and emphasise a focus on users, or actors, to inspire reasonable interaction. In this way essential use cases can provide an effect similar to the early development of user interface prototypes as an input to system design. In this paper we outlined our approach to using essential use cases with cards and roleplay for requirements analysis and review.

Other advantages concern system design based on the use cases. The system responsibilities can also be extended to include responsibilities that go beyond interaction with the user, and so document important contextual functionality. The responsibilities determined can also be used as a starting point for design by distribution of system responsibilities. In this paper we outlined our approach to using this distribution of responsibilities to develop an object-oriented design directly from the essential use cases, and we showed a detailed example. The approach offers the advantage of traceability, plus some operation guidance in design.

References

Beck, K. and Cunningham, W.: 1989, A laboratory for teaching object-oriented thinking, *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pp. 1–6.

Bellin, D. and Suchman Simone, S.: 1997, *The CRC Card Book*, Addison-Wesley.

Biddle, R., Noble, J. and Tempero, E.: 2001, Role-play and use case cards for requirements review, *in* D. Cecez-Kecmanovic, B. Lo and G. Pervan (eds), *Proceedings of the Australasian Conference on Information Systems (ACIS2001)*, Coff's Harbor, NSW, Australia.

Biddle, R., Noble, J. and Tempero, E.: 2002a, Essential use cases and responsibility in object-oriented development, *in* M. Oudshoorn (ed.), *Proceedings of the Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia.

Biddle, R., Noble, J. and Tempero, E.: 2002b, Sokoban: A system object case study, *in* J. Noble and J. Potter (eds), *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS-Pacific)*, Sydney, Australia.

Biddle, R., Noble, J. and Tempero, E.: 2002c, Supporting reusable use cases, *in* C. Gacek (ed.), *Software Reuse: Methods, Techniques, and Tools, 7th International Conference, ICSR-7, Austin, TX, USA, April 15-19, 2002, Proceedings*, Vol. 2319 of *Lecture Notes in Computer Science*, Springer.

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

Jacobson, I., Booch, G. and Rumbaugh, J.: 1999, *The Unified Software Development Process*, Addison-Wesley.

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

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

Wirfs-Brock, R. and Wilkerson, B.: 1989, Object-oriented design: A responsibility-driven approach, *in* N. Meyrowitz (ed.), *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pp. 71–75.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L.: 1990, *Designing Object Oriented Software*, Prentice Hall.