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

School of Mathematical and Computing Sciences
Computer Science

Sokoban: A System Object Case Study

Robert Biddle, James Noble, Ewan Tempero

Technical Report CS-TR-01/11
September 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

Sokoban: A System Object Case Study

Robert Biddle, James Noble, Ewan Tempero

Technical Report CS-TR-01/11
September 2001

Abstract

This report presents a case study applying the *Distribution of System Responsibilities* (DSR) object-oriented design technique. DSR provides a starting point for an object-oriented design by directly manipulating the requirements for that application. The case study applies DSR to the development of a small application, the game Sokoban. The case study provides evidence that DSR does allow object-oriented designs to be created, that it allows reasonable designs to be created, and that any object-oriented design can be created by it.

1 Introduction

This report presents a case study applying the *Distribution of System Responsibilities* (DSR) object-oriented design technique [2]. DSR provides a starting point for an object-oriented design by directly manipulating the requirements for that application. This has the advantage of providing operational guidance for doing design, and also allows traceability between the requirements and design to be easily determined.

The purpose of the case study is a first attempt at determining whether or not DSR has any validity. The specific questions we hope to answer are:

- Can *any* object-oriented design be created using DSR? That is, does DSR actually produce a design, or does what it produces require more work before a design is available.
- Can a *reasonable* object-oriented design be created using DSR? That is, if it does produce a design, is it a sensible one, or does it have an obvious flaw (such as being essentially a functional decomposition, which is often not a reasonable solution).

Another question that must be answered at some stage, but which will not be directly addressed by this case study is:

- How many reasonable object-oriented designs will DSR lead to? It is often the case that there are several reasonable designs for the system, and the most appropriate depends on the non-functional requirements. If DSR doesn't allow all designs, then it may be that it will not allow the best solution to be found.

The application used for this case study is the Sokoban Game (see section 3). This is a small application, so the result of this case study will not be conclusive, but it provides a useful starting point.

The report is organised as follows. The next section provides an outline of the method used in the case study, including an overview of DSR. Section 3 describes and discusses the Sokoban game being implemented. The first step in DSR is to describe the requirements of the proposed system as a set of essential use cases [6]. This is done in section 4. Section 5 shows a series of iterations of an object-oriented design for the Sokoban system, beginning with the *system object* created from the essential use cases in section 4. We then reflect on the process and the outcome of the case study in section 6, including evaluating the design that we have developed and examining the traceability that DSR provides between the requirements and design. Finally, We summarise and present our conclusions.

2 Method

The method used in this case study is to describe the application (the Sokoban game), apply DSR, and then reflect on both the resulting design, and the process itself.

DSR is supposed to produce an object-oriented design for a system. It begins by identifying the *essential use cases* (EUCs) for the application under consideration. EUCs are a variant of use cases [8] that were developed by Constantine and Lockwood as part *Usage-Centered Design*, a technique for user interface development [6]. EUCs are presented as *user intentions*, a description of what the user wants to achieve, and *system responsibilities*, what the system is expected to do in response. We have adapted EUCs for full software development by requiring that the description of the system responsibilities must also include responsibilities whose effects would not normally be visible.

Any object-oriented design can be presented as a set of CRC cards [1]. A CRC card for a class has the name of the class, its "responsibilities" — what other parts of the design expect from it, and its "collaborators" — the other classes of the objects that it must communicate with in order to carry out its responsibilities. The key observation that underpins DSR is the fact that there has to be some relationship between the system "responsibilities" of the EUCs, and the class "responsibilities" in any design for the system. Uncovering this relationship is the what DSR is all about.

DSR "discovers" the relationship between the system and class responsibilities by creating it. Beginning with the complete system responsibilities — the *system object* — the designer makes decisions as to how to *distribute* those responsibilities amongst different classes. The relationship between system and class responsibilities is then the set of all the decisions. By tracking the decisions, this relationship can be made explicit, thus providing *traceability* between the requirements and the design.

The design in this case study will be presented as a sequence of *iterations*. While not specifically part of DSR, this provides a useful mechanism for examining the decisions made about distributing responsibilities for the purposes of evaluating the case study. In fact, we generally expect that DSR will be used in an iterative and incremental process, so dealing with iterations will be the typical case.

Exactly what determines an "iteration" is a matter for debate, and not discussed here. What was done for this case study was to make a decision about a class (for example, creating a new class, or changing the intent of an existing class), and the distribute the responsibilities to match the decision.

DSR is still under development, and what is learned in this case study will aid in its development. Two issues to be resolved is how to determine when a new class is needed in the design, and how to distribute the responsibilities between the classes. For the purposes of this case study, the solutions to these issues are "whatever seems sensible". While not entirely satisfactory, this will allow us to make progress on the case study, and we leave better solutions to these issues to future work.

3 Sokoban

The Sokoban game 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 either the location beside the box opposite from the worker is either empty, or contains a 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 1 shows what a solution may look like.

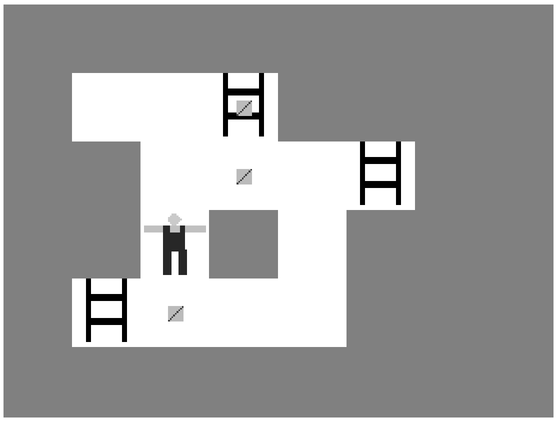

Figure 1: An example of what a Sokoban level might look like. Shown is the worker, three shelves, and three boxes. One of the boxes is on a shelf.

4 Requirements

This section describes the requirements for the Sokoban system as a set of essential use cases (EUCs). The convention with EUCs it 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 "choose operation", as that is implied by the EUC name. Similarly, the two column format specifies which participant does which action [7]. 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).

Start Game	
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

Restart Level	
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

Open Level	
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

Move Worker	
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

Show Hall of Fame	
User intention	System Responsibility
	• List all players in the Hall of Fame

5 Creating an Object-Oriented Design

This section presents a series of iterations an object-oriented design for the Sokoban system. The responsibilities in each class are labelled so that traceability between the requirements and the design can be tracked [9]. This will be discussed further in section 6.3, 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).

5.1 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, as there are obviously no collaborators for this class.

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

5.2 Iteration 2

Examining the System Object in section 5.1, 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)	**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.

5.3 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.

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)	**Location**

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	

5.4 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)	• **Level** • **Player**

5.5 Iteration 5

Finally, we create a **HallOfFame** class.

Sokoban	
S1 If all the boxes in the Level are on a shelf, then load the next Level (to LEV1) **SO2.1** Given a direction, move the Worker in that direction. (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 (to P3, P4, HOF1, HOF2, HOF3) **SO4** If the current level is any level other than the first one, record that the player cannot enter the Hall of Fame (to P2) **SO5** List all players in the Hall of Fame (to HOF1,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 (to LEV2) **SO12** Show the current Level (to LEV3)	• **Level** • **Player** • **HallOfFame**

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)	

5.6 The Final Design

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 [10].

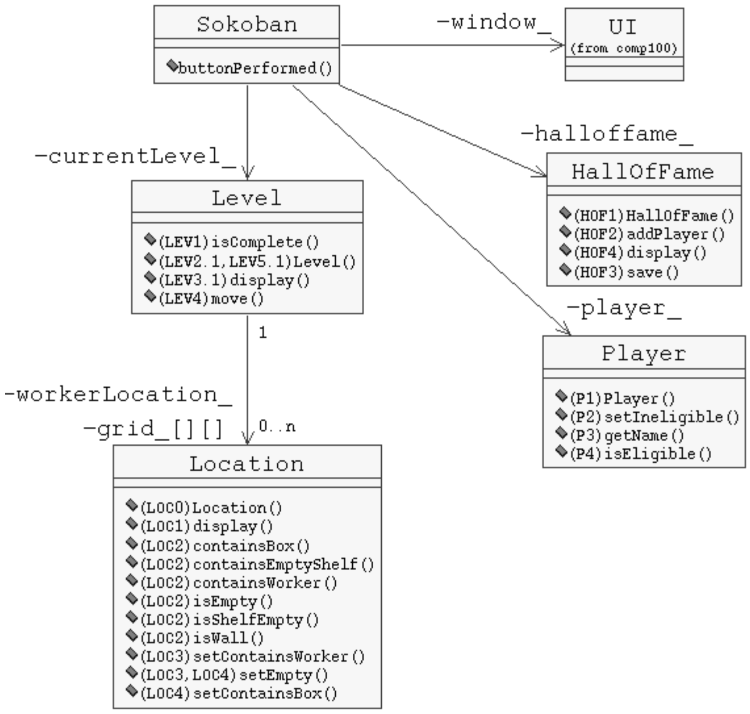

Figure 2: UML reverse-engineered from a Java implementation of Sokoban.

6 Discussion

6.1 Evaluation

Based on our experience in object-oriented design, the final design we have produced looks fairly reasonable. In fact, we have implemented this design, as shown in figure 1. Figure 2 shows a class diagram developed from a model created by Rational Rose's reverse engineering facility applied to a Java implementation of this design. There are some minor other differences that result from implementation choices, such as the use of the `UI` class. The `UI` class provides a very simple user interface, with the user actions indicated by pressing buttons. The method `buttonPerformed` handles all the **Sokoban** responsibilities by dispatching to the appropriate private methods.

The design was straightforward to implement. Perhaps the most complex decision that had to be made was how to represent the relationships between **Location**s. The ease with which the design was implemented suggests that the design is not unreasonable.

6.2 Relationship to other designs

At the time this case study was done, there were already several local implementations of Sokoban, some by one of the authors (Tempero). Two of the designs are shown in figures 3 and 4. Neither of these designs implement the "Hall of Fame" functionality (and hence also don't need to know about "players"). The first design was in fact developed from an existing Pascal implementation of Sokoban. The second design was one of several attempts to "improve" the first design.

The fact that designs already existed before the case study began raises questions as to how much they influenced the decisions made about how to distribute responsibilities. In fact, the implementation shown in figure 2 used some of the code from both of the earlier

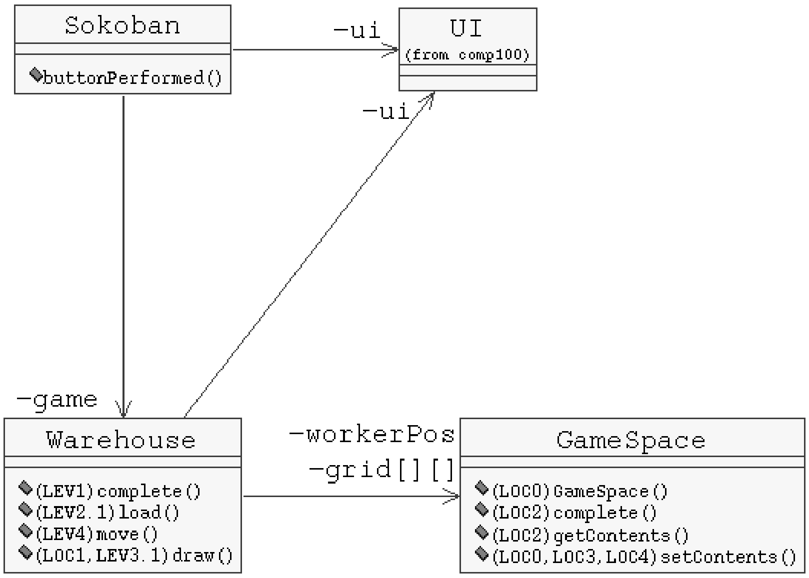

Figure 3: The design for the our first object-oriented implementation for Sokoban

implementations! Nevertheless, the design developed here is different from the designs used in the earlier implementations, as can be seen by the different distribution of responsibilities. This gives us some confidence that the design developed for the case study was at least in part the result of using DSR, rather than completely a consequence of previous experience.

Note also that it is fairly clear how the distribution of responsibilities in the design from this case study relates to the distribution of responsibilities in the earlier designs. This is very encouraging, as it suggests that DSR is fairly robust with respect to the kinds of designs it can lead to. In fact, we believe that any object-oriented design can result from the application of DSR, and this evidence supports that belief.

6.3 Traceability

One of the claimed major advantages of DSR to object-oriented design is that traceability between requirements and design can be easily derived. Figure 5 shows how the responsibilities in the System Object (which came from the requirements) can be traced to responsibilities in the other classes in the design.

With this diagram, we can determine what parts of the design would be affected by changes to the requirements. For example, suppose we decided to change the requirements so that the player's name is only asked for when the game is finished, rather than when it begins. This would the system responsibilities for dealing with the player's name in the **Start Game** use case, and moving them to the **Move Worker** use case. This would in turn affect responsibilities **SO3**, **SO7**, and **SO8**, and, from figure 5, we can see which classes might be affected by these changes.

6.4 Reflecting on the process

This case study does raise further questions. Perhaps the most important is:

- Does the way we write the essential use cases affect the quality of the design?

Given the general uncertainty surrounding the definition of a use case (or rather, exactly what the definitions that have been given actually mean), it seems reasonable that different

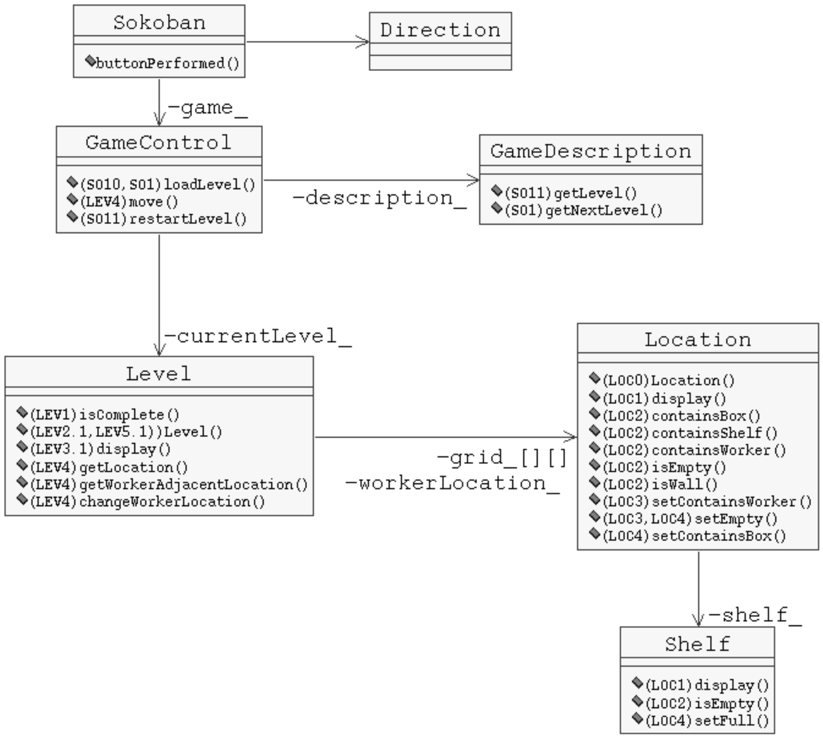

Figure 4: A second design for Sokoban, completed before the case study.

people may come up with use cases different from ones we gave in section 4, or would represent the responsibilities differently than we did. Presumably this might lead to a different design.

The most common question about the definition of use cases seems to be "how big" a use case should be. Cockburn, for example, has (in essence) three different sizes [5], and other commentaries seem to suggest others use different sizes as well. The heuristic we use when deciding "how big" a use case is:

> What do we expect to get from a "single interaction" with the system to be built?

In the case of the Sokoban game, we have seen uses cases suggested such as "Play a level", or even "Play a game". These are clearly both reasonable goals for the player, and since use cases are supposed to be associated with a goal of the user, they seem to be reasonable choices. However, from the point of view of understanding the requirements of the system, they do not provide much insight as to what the system may look like, only what it will be used for. The use cases we give provide a clear view (we believe) of what the system will look like.

One concern with this heuristic is that is seems to imply a specific user interface. For example, when one uses a keyboard to enter text, the heuristic would suggest that "enter 1 character" is a use case. However, if interactive voice recognition was used to enter text, then that use case clearly would not make sense.

Related to the issue of the expected user interface is the meaning of "single interaction". When someone is filling out a form in a web browser, is the typing in the text areas covered by one (or more) use case, or is it the submission of the form? The dialog boxes used in many applications have a similar issue, particularly as some of them can be "smart", that is, providing application specific functionality before any buttons on the dialog box are selected.

We speculate that this heuristic is close to being what is really needed, at least for DSR to work. Other sizes of use cases may be useful for other purposes relating to requirements

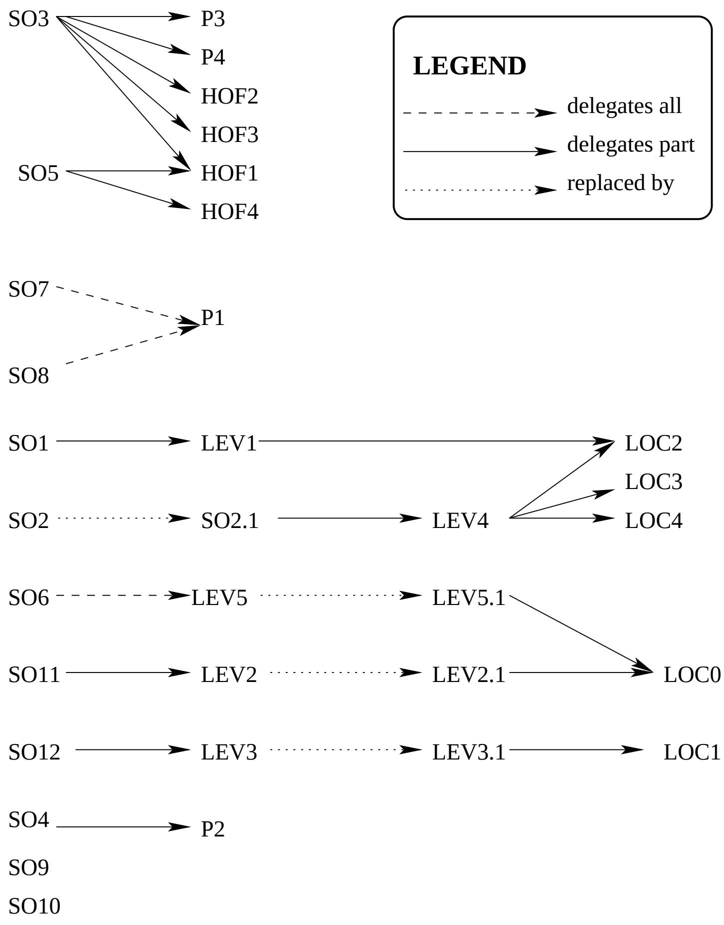

Figure 5: The traceability between the class responsibilities

elicitation and reporting (which is what Cockburn uses the different sizes for). Our experience with students is that they quickly pick up the "right" level, so we do not believe this is a big issue.

Related to the question about use cases are the following questions:

1. What is a responsibility?

   The motivation for this question comes from use cases such as **Move Worker**. The presentation of this use case given in section 4 lists four responsibilities. These prompt the following questions:

   (a) Why were these divided up into four responsibilities? Why not a different number? In particular, why not just combine them all into one responsibility?

   Notice that one of these responsibilities, *Show the current Level*, appears in other use cases. This is a good argument for dividing responsibilities up in the way done in this use case, but whether it makes sense to do so does seem to depend on how the responsibility is written (see the discussion for question 2). And it still doesn't answer the question of why "four", and not some other number.

   (b) Three of these responsibilities start with "If". What does a conditional like this mean in a responsibility?

   Cockburn warns against the use of conditions in use cases, but he seems to be more concerned with the complicated control flow that can result. Our use of the condition here is more as a guard. Is this distinction important? Or does Cockburn's concern still apply? It's hard to imagine how to write this use case without conditionals (but perhaps that's a fault with the choice of use case?).

2. How should a responsibility be represented?

   As discussed in question 1a, what gets recorded as a responsibility depends on how it is written. We could have written the system responsibility in **Move Worker** as one big responsibility, rather than the four given.

   The use of the "guard" conditional may prove important to deciding representation. Notice that the one responsibility not "guarded" is *Show the current Level*. Furthermore, it seems correct that it not be guarded — it happens whether or not the worker moves in the first responsibility. Of course it also may be correct to not *Show the current Level* if the worker doesn't move, and yet because the responsibility appears in other use cases, we almost certainly want to separate it out.

   One possibility is to just leave it as one responsibility, and allow it to sort itself out when the transformation of responsibilities take place. For example, *Show the current Level* should not really be a System Object responsibility, but just a responsibility on one of the classes in the resulting design (**Level** in this case). This seems to neatly avoid many concerns about how writing down the System responsibilities affects the design, although possibly this just pushes the problem off to the distribution/transformation of responsibilities.

   The first decision we discussed (section 5.2) was based on the use of the term "Level". It is likely that our use of "Level" in the use cases resulted from the carefully worded description in section 3.

   In fact, the form of the use cases presented in this case study is not the original form that they were written. The original form used inconsistent wording for responsibilities in different use cases (for example "Show the resulting level" in **Move Worker**). We spent some time making sure that consistent phrasing was used so that the System Object would not contain extraneous responsibilities. It is possible that our choice of terminology biased the resulting design.

   This suggests that for this approach to be successful, there needs to be some way to use consistent terminology and phrasing for the responsibilities. One way to achieve this might be to use patterns in the development of the EUCs [3].

3. How are responsibilities distributed?

   Ultimately, the quality of the resulting design depends on how responsibilities are distributed and transformed. Moving a responsibility from one class to another does not seem to pose any problems, but how responsibilities are transformed seems crucial. Clearly there is a lot of work to be done here in understanding what is possible, and what the consequences are. We will need a lot more experience with this approach before we can offer anything concrete.

   We do, however, make one observation. A technique that is often used to determine classes is text analysis (see [4] for example). This analysis is typically done on documents relating to the system being developed. What we were effectively doing in this case study was to use text analysis on the *responsibilities*. Whether this observation has any significance remains to be seen.

6.5 Evaluating the case study

As mentioned in section 2, it is not the purpose of this case study to "prove" anything about DSR, but just to provide evidence that it has some potential to be useful. Nevertheless, it is worth evaluating how convincing the evidence is, in order to plan future case studies. Our evaluation takes the form of identifying issues that arise in the case study.

1. The original description of Sokoban may provide guidance to creating a design, and so influence the choice and composition of the essential use cases.

2. The essential use cases were carefully written (sometimes multiple times). It may be that had they been written differently, then either the process may not have gone so smoothly, or the resulting design may not have been so reasonable.

3. All of the essential use cases were available from the start. For larger systems, particularly those developed with an iterative and incremental process, early design will often be done with only some of the use cases available.

4. There were already existing designs (and implementations) for Sokoban. These may have guided the decision about the distribution of responsibilities.

5. We claim that the design we developed using DSR was reasonable, but made no formal attempt to evaluate it. We need to establish whether DSR helps or hinders the development of good designs.

6. The Sokoban game is very small. DSR may not work so well with a larger application.

7 Conclusions and Future Work

This report has presented a case study looking at how an object-oriented design can be developed from essential use cases and distribution of responsibilities — DSR. With the number of provisos we have noted throughout the text, it will come as no surprise that the result of the case study is not absolutely conclusive. However, we believe it does provide evidence that DSR does allow object-oriented designs to be created, that it allows reasonable designs to be created, and that any object-oriented design can be created by it — these were the questions we identified in the introduction. From this, we believe that is is worth proceeding with a larger case study.

There are in fact several possibles for the next case study. It would be worth while performing a case study of a similar size as this one but involving a range of people. Something the size of Sokoban can be easily done in an afternoon, and this would provide useful feedback on our questions about the importance of choice of use cases and responsibilities.

However, the real test will be with larger systems. Even with Sokoban, managing the responsibilities and their distribution is tedious, and that's only five use cases. A system that requires 1-3 person months might have 60-100 use cases — clearly good tool support will be needed before this approach will be at all viable.

8 Acknowledgements

Duncan Baily was responsible for first Java implementation of Sokoban, and hence the look and feel shown in figure 1.

References

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

[2] Robert Biddle, James Noble, and Ewan Tempero. Essential use cases and responsibility in object-oriented development. Technical report, Victoria University of Wellington, 2001.

[3] Robert Biddle, James Noble, and Ewan Tempero. Patterns for essential use cases. In *KoalaPLoP*, Melbourne, Australia, 2001.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, second edition, 1994.

[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] Larry L. Constantine and Lucy A. D. Lockwood. *Object Modeling and User Interface Design*, chapter Structure and Style in Use Cases for User Interface Design. Addison-Wesley, 2001.

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

[9] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice.* Prentice Hall, 1998.

[10] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In Norman Meyrowitz, editor, *Proc. of OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pages 71–75, 1989.