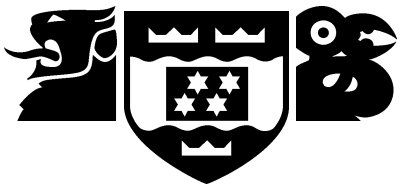
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

Tool support for traceability between requirements and design

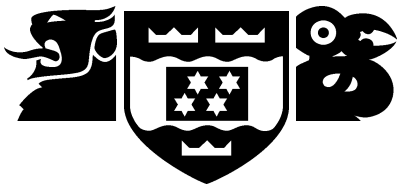
Robert Biddle, James Noble, Ewan Tempero[1]

Technical Report CS-TR-02/7
June 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

Tool support for traceability between requirements and design

Robert Biddle, James Noble, Ewan Tempero[2]

Technical Report CS-TR-02/7
June 2002

Abstract

Traceability is the ability to determine relationships between different development artifacts in the software development process. Being able to automatically trace from the requirements through to the implementation allows questions such as "Are all the requirements met by the implementation" to be reliably answered. While answers to such questions are important for managing the quality of both the process and the product, gathering and managing the relationships that allows such automatic checking can be expensive, making it less clear that the benefit justifies the cost. The creative nature of design means that the relationships between requirements and design cannot be derived automatically, and so their management cost is high. We have developed a design technique that, with appropriate tool support, can significantly reduce that cost. We present Ukase, a web-based tool that provides such support.

Keywords: web-based tool support, traceability, object-oriented design

1 Introduction

Traceability is the ability to determine relationships between different development artifacts in the software development process. Traceability is used to manage quality. By knowing the relationship between artifacts, questions such as "Are all the requirements met by the implementation?" can be answered by tracing through the relationships. If the relationships are represented explicitly, then the tracing can be done automatically. In this paper, we present **Ukase**[1], a web-based tool that supports traceability between the requirements and the design of a system under development.

Maintaining traceability relationships is a non-trivial task. Some relationships can be deduced automatically (for example, implementation stubs can be automatically generated from the design), which means that the cost associated with maintaining traceability can be minimised. However, other relationships, by their very nature, require human intervention. An example of this is the relationship between requirements and design. The design process requires creative decisions to be made, and so the relationship between requirements and design can only be supplied by the designer. If the tool support for traceability requires the user to manually manage the relationships, then it significantly increases the cost of the design process. Furthermore, it introduces the likelihood of inconsistency between the actual relationships and those maintained by the tool. This reduces the benefits of maintaining traceability in the first place. This means that the best form of tool support must be part of the design process.

The main goal of Ukase is not in fact to manage traceability, but to support a technique for developing object-oriented designs from use cases. This technique, *Distribution of System Responsibilities* (DSR), produces the traceability relationships as a natural outcome, rather than needing them to be determined as a separate activity. As a consequence, Ukase has all the information it needs to manage traceability, without the designer having to do extra work.

The paper is organised as follows. In the next section we discuss the background to this work, in particular traceability and DSR. In section 3, we present Ukase in detail. We then discuss our experience using Ukase in section 4, and following that, we compare Ukase with other work in the literature. Finally we present our conclusions.

2 Background

2.1 Essential Use Cases

Ukase represents the functional requirements as a particular form of use cases known as *essential use cases*. Jacobson et al. define a use case in their 1992 book as "a behaviorally related sequence of transactions in a dialogue with the system"[9]. A more recent definition for the Rational Unified Process (RUP) and the Unified Modeling Language (UML) shows little real change, saying a use case is "a description of a set or sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor"[8]. The general idea of a use case is to represent intended sequences of interaction between a system (even if not yet implemented) and the world outside that system.

Essential use cases were designed to avoid built-in assumptions, often hidden or implicit, about the form of the user interface expected by the RUP style use cases. The term "essential" refers to essential models that "are intended to capture the essence of problems through technology-free, idealized, and abstract descriptions". Essential use cases are documented

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

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 1: 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.)

in a format representing a dialogue between the user and the system. This resembles a two-column format used by Wirfs-Brock [19].

Constantine and Lockwood give the examples shown in Figure 1. The narrative in the use case on the left is for a conventional use case, described in terms of actions and responses. The dialogue in the use case on the right 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.

2.2 Traceability

Traceability refers to the ability to determine relationships between different development artifacts in the software development process. Having this ability allows useful questions about the quality of both the process and the artifacts to be answered. Such questions include: "Do the requirements satisfy the needs of the client?", "Does the implementation satisfy the requirements?", "If we change this requirement, how does that impact the design and implementation?" This makes traceability a powerful tool for quality control and improvement.

As indicated by the questions above, traceability typically deals with the requirements of the system under development. An important part of requirements engineering is requirements traceability, which Gotel and Finkelstein define as "the ability to describe and follow the life of a requirements, in both a forwards and backwards direction" [7]. They also distinguish "pre-RS" traceability — those aspects of a requirement's life prior to its inclusion in the system specification — from "post-RS" traceability — those aspects of its life after its inclusion. This view of traceability focuses on relationships between artifacts in different

phases of the life-cycle of the system, what Pfleeger refers to as *horizontal traceability* [14]. Pfleeger also identifies *vertical traceability* as relationships between artifacts in the same phase of the life-cycle.

Traceability is a form of documentation, as opposed to an inherent part of the delivered system — it is possible to deliver a working system without any explicit record of traceability just as it is possible to deliver a system without any documentation. Whether this is desirable is still open to debate. While there is general agreement that using traceability is useful for managing quality, there is evidence that there can be significant costs involved [11]. Part of the motivation for our research is to investigate ways to maintain traceability but at minimal cost. As discussed in the introduction, our goal is to provide support for requirements to design traceability at minimal cost to the designer.

2.3 Distribution of System Responsibilities

Distribution of System Responsibilities (DSR) is a technique for developing an object-oriented design from requirements expressed as use cases. DSR assumes the requirements describe a set of *responsibilities* that the system under development must fulfill. An object-oriented design of a system can also be thought of as sets of responsibilities (each set belonging to a class) collaborating to meet the system responsibilities. This is the view presented by the Class-Responsibility-Collaborator (CRC) design technique [1].

The starting point for DSR is to gather all of the system responsibilities identified in the EUCs together and treat them as *class* responsibilities for a (potentially fairly large!) class. If there were such a class that had such responsibilities, then it would represent an object-oriented design (albeit not a very good one) that satisfied the requirements of the system under development. There would only ever need to be one instance of this "system" class, and so we refer to this design as the *system object*.

The observation on which DSR is based is that, given an object-oriented design (with more than one class), there has to be a relationship between the system responsibilities and the sets of class responsibilities in the design. This means that the design process is about *distributing* the system responsibilities among the classes. DSR advocates thinking of the design process as exactly making decisions about how to do the distribution. A consequence of DSR is that every decision about why a class has a particular responsibility is explicitly tied to supporting a particular requirement (system responsibility). The relationship between the class responsibility and the system responsibility is exactly the horizontal traceability between requirements and design.

As well as relationships between the system responsibilities and class relationships, there are also relationships between class responsibilities. In order for one class to discharge one of its responsibilities, it may make a request of, or *delegate* part of that responsibility to, another class. Establishing that such delegation should take place can also be thought of as distributing responsibilities, and so DSR can be used to flesh out a complete design by specifying inter-class relationships, or, vertical traceability. The intermediate and final products of the DSR process are an object-oriented design, typically presented as a set of CRC cards.

In practice, decisions about the distribution of responsibilities depend on the domain model, and decisions made about software architecture, and use of design patterns and/or frameworks. Further details are available elsewhere [2, 3].

Ukase supports DSR by first supporting the capture of requirements as EUCs, and then by supporting the process of deciding how to distribute the system responsibilities identified in the EUCs among the classes that make up the design. While the user only has to specify how the distribution is to take place, Ukase captures each distribution decision, and so records the traceability relationships without any extra cost to the user.

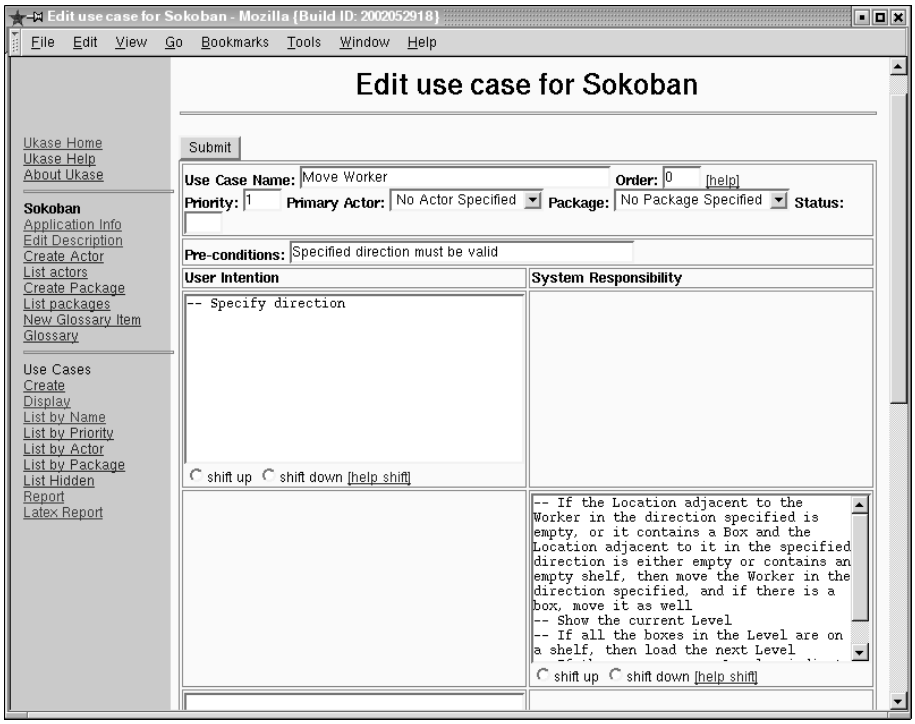


Figure 2: Entering the details of a use case in Ukase.

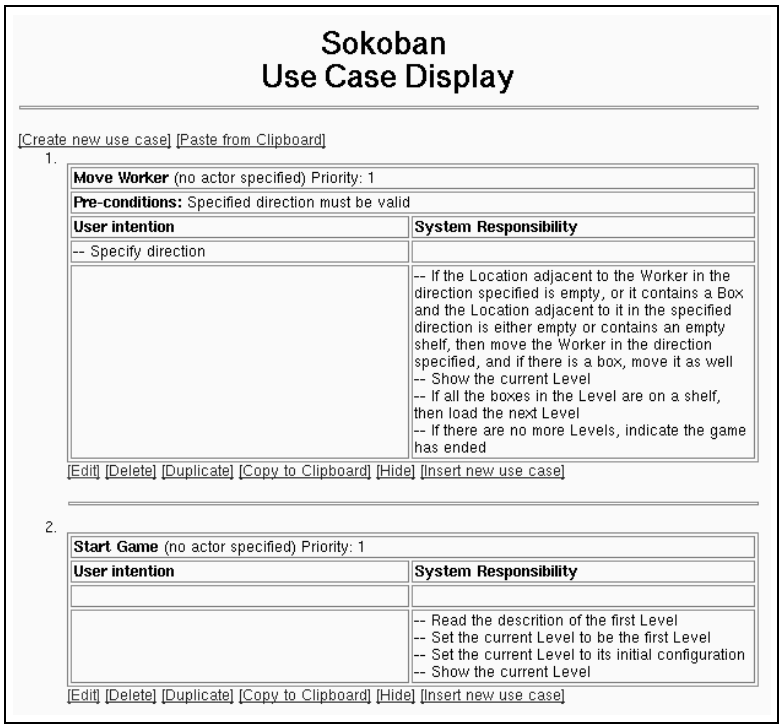
2.4 Running Example

We will use a running example to demonstrate the features of Ukase. The example is the Sokoban game, which we have used for a case study of DSR [3]. 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 we use here is slightly simpler than the one used for the case study, namely:

> 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.

3 Ukase

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 [18] CGI script [6], with the information stored in a MySQL database [12]. It is highly accessible (anywhere on the Internet), and may be used as groupware (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.



Sokoban
Use Case Display

[Create new use case] [Paste from Clipboard]

1.

Move Worker (no actor specified) Priority: 1	
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 there are no more Levels, indicate the game has ended

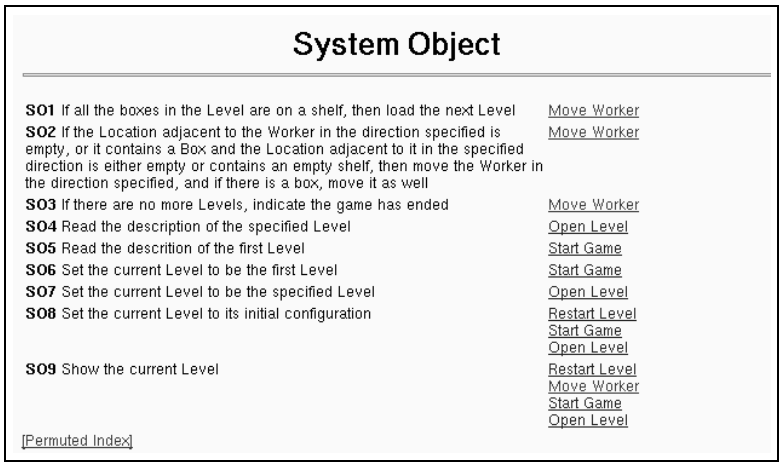
[Edit] [Delete] [Duplicate] [Copy to Clipboard] [Hide] [Insert new use case]

2.

Start Game (no actor specified) Priority: 1	
User intention	**System Responsibility**
	-- Read the descrition of the first Level -- Set the current Level to be the first Level -- Set the current Level to its initial configuration -- Show the current Level

[Edit] [Delete] [Duplicate] [Copy to Clipboard] [Hide] [Insert new use case]

Figure 3: Some of the use cases for Sokoban.



System Object

SO1 If all the boxes in the Level are on a shelf, then load the next Level	Move Worker
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	Move Worker
SO3 If there are no more Levels, indicate the game has ended	Move Worker
SO4 Read the description of the specified Level	Open Level
SO5 Read the descrition of the first Level	Start Game
SO6 Set the current Level to be the first Level	Start Game
SO7 Set the current Level to be the specified Level	Open Level
SO8 Set the current Level to its initial configuration	Restart Level Start Game Open Level
SO9 Show the current Level	Restart Level Move Worker Start Game Open Level

[Permuted Index]

Figure 4: The System object for Sokoban

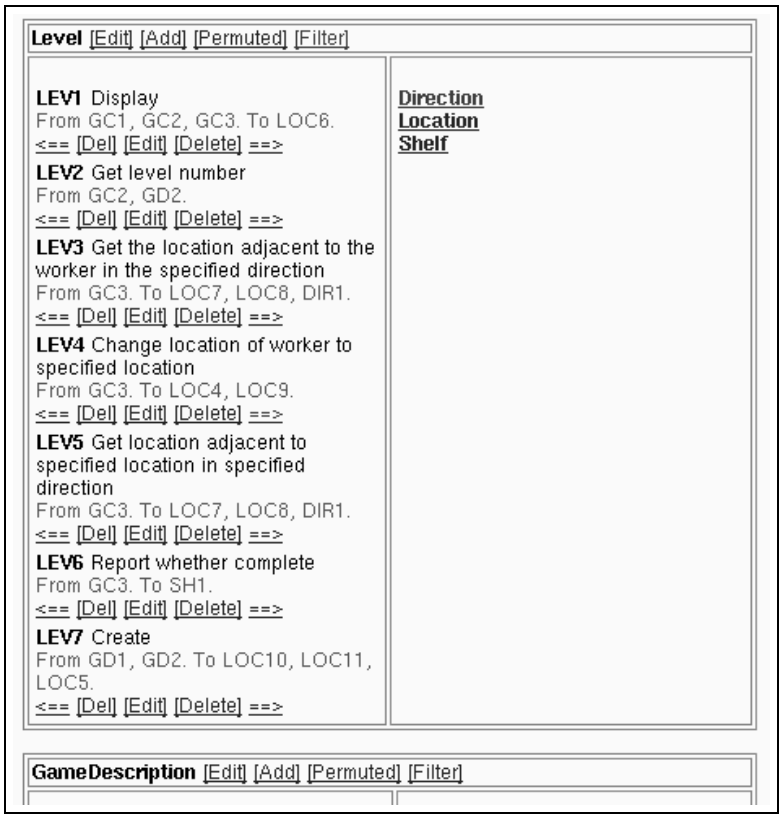


Figure 5: The **Level** class in a design for Sokoban.

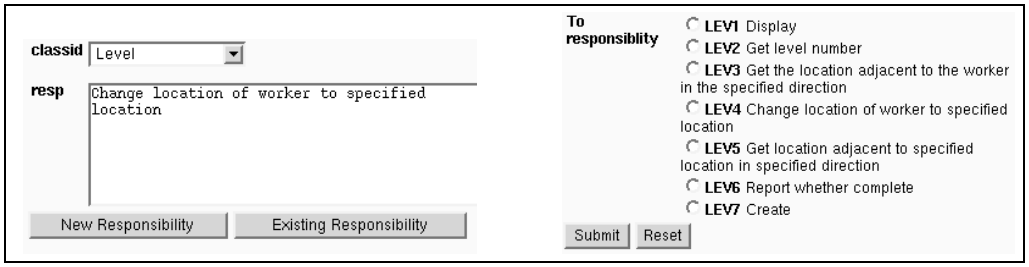


Figure 6: Two forms for delegating responsibilities to the **Level** class.

3.1 Supporting Use Cases

Ukase provides tool support for DSR, and so a significant part of its functionality is management of use cases. Much of this functionality has been discussed elsewhere [4], but we summarise it here.

Ukase provides a mechanism for entering the details of the essential use cases, and a mechanism for displaying the results. The input is done using a HTML form laid out the way that a use case would look in a report (see Figure 2). 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 3, and once development is complete, the resulting model is available as a report. Use cases can also be listed in a variety of ways (grouped by package, by actor, and so on).

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 detailed explanations of such terms can be recorded. 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.

3.2 Supporting Responsibilities

In order to support DSR, Ukase has to manage the requirements at the level of responsibilities. This is done by the requirements engineer following the convention of using "`--`" at the beginning of a line to indicate a responsibility (see Figure 2). We chose to do it this way because it provided a lightweight mechanism for specifying responsibilities. Alternatives that used separate fields for each responsibility would have raised the issue of how to specify a variable number of responsibilities. While solutions to this exist, they all add to the complexity of the user interface. The use of a simple convention is a good compromise.

The responsibilities on the system side of a use case are what are used to create the system object. Figure 4 shows the system object for the requirements for the Sokoban system. On the left are the responsibilities that come from the use cases, and on the right are links to the use cases the responsibilities come from. Note that some responsibilities may come from more than one use case.

Ukase represents a design essentially as a set of CRC cards [1]. Each class in the design is represented as a card, showing the class name, responsibilities, and collaborators (other classes with which it cooperates). Figure 5 shows one such "card" for the **Level** class. This is part of a web page that lists all of the classes currently in the design (the **GameDescription** class can be seen beginning at the bottom of the figure). It provides controls (beside the class name) for managing details of the class as a whole.

There are also controls for managing details of individual responsibilities. These operations include changing or deleting responsibilities from a class, as well as distributing responsibilities and tracing between responsibilities (see below).

3.3 Supporting DSR

Ukase directly supports DSR by including mechanisms for distributing responsibilities. This control (labelled `[Del]` in figure 5) is used to allow the delegator class to delegate part (or all)

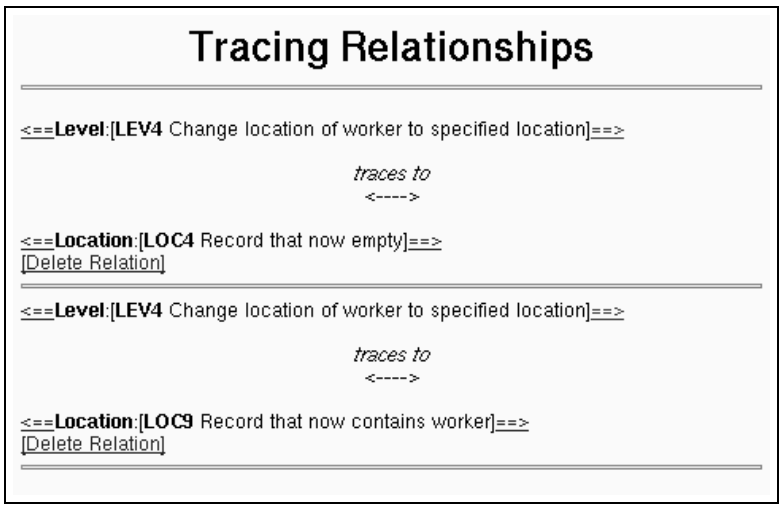


Figure 7: The responsibilities that **LEV4** delegates to.

of the specified responsibility to another class. Clicking the `[Del]` control produces a form for specifying the responsibility to be delegated to, and the class that responsibility belongs to (see the left side of Figure 6). If the other class already has the needed responsibility (typically because of an earlier design step) an existing responsibility can be chosen (right side of Figure 6). Whichever distribution choice is made, the traceability relationship between the delegating and delegatee responsibilities is automatically recorded.

3.4 Supporting Traceability

The view of a class, in particular the view of a responsibility within a class, shows what is currently known about relationships between responsibilities. Immediately above the controls for a responsibility is a summary of the relationships. For example, in Figure 5, for the **LEV4** responsibility, "`From GC3. To LOC4, LOC9.`" indicates which responsibilities delegate to it (`GC3` in this case), and which it delegates to (`LOC4` and `LOC9`). The labels are used as short-hand for the responsibilities and are user-specified (the number is automatically generated, but can be changed by the user). The arrows (`<==` and `==>`) show the details of the relationships. Figure 7 shows the result of following the `==>` link for **LEV4** in Figure 5. The user can navigate through the design, following the chain of delegation forward or backward to the originating use case.

3.5 Other support

As discussed in section 2, the design process consists of making decisions about where to distribute responsibilities. It is crucial that good support be provided to help with this decision making process. *Text analysis* is a technique that is advocated by many as a way to identify classes [5]. A primitive, but surprisingly effective form of text analysis is to compute a permuted index (also known as key word in context) of the responsibilities. Figure 8 shows part of the permuted index for all the responsibilities in the system object for Sokoban. Each entry in the index links back to the originating use case. Ukase can also show the permuted index for responsibilities in a specified class. The permuted index can present the user with more information than can be usefully managed, and so Ukase also provides a simple "filter" facility, which shows only those responsibilities that match the filter.

Ukase supports the design process by considering a design as a sequence of "iterations". An iteration is whatever the designer wants it to be, but the intent is that each iteration is a

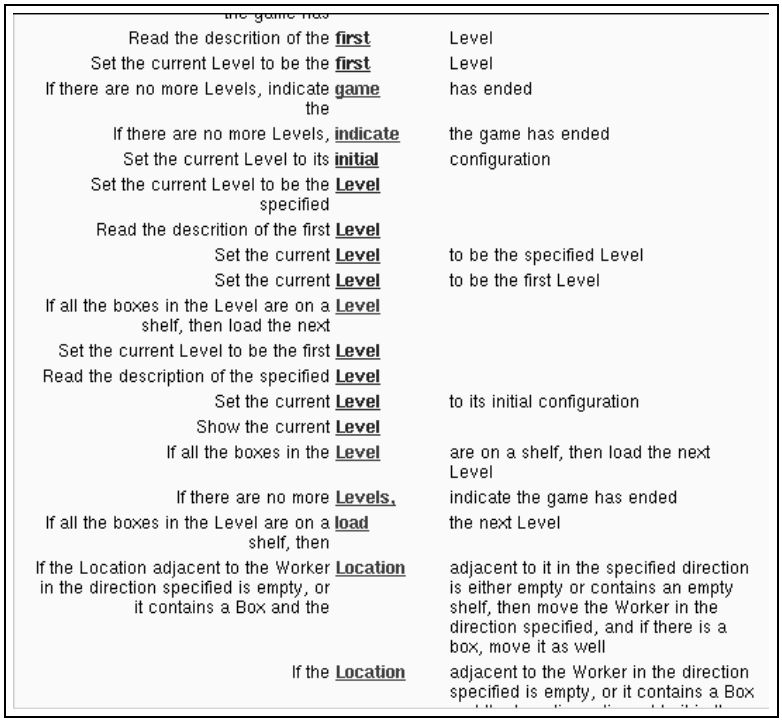


Figure 8: Part of the permuted index for the system responsibilities for Sokoban

more refined design than the previous iteration. An iteration represents a checkpoint in the design process, from which different design choices can be explored. The first iteration is the system object.

4 Discussion

The first version of Ukase was first in use about August 2000. This version only provided management of use cases. This aspect of Ukase been used in several research projects, a commercial case study, and in several courses (final year undergraduate and graduate). It has also undergone heuristic evaluations [13] for usability, and the students in one of the graduate courses critiqued its usefulness 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.

The support for distribution of responsibilities has been developed since August of 2001. This aspect of Ukase has only been used by the authors, but has been used to develop implementations for Sokoban (about 600 lines of Java) as well as two other applications of about 3000 lines of Java each (and 3000-4000 lines of Java generated from JavaServer Pages files). This experience indicates that, although the tool is still being developed, it does provide effective support for the design process.

Ukase supports both horizontal and vertical traceability. Horizontal, because it links the use cases (requirements) to the design, and vertical, because it shows the relationships between different parts of the design. Having the ability to quickly navigate around the relationships within the design and back to the requirements gives confidence in the design being developed.

The main addition we want to make to Ukase is to provide visual support for DSR. As with any other design technique, it is difficult to keep track of the current status of the

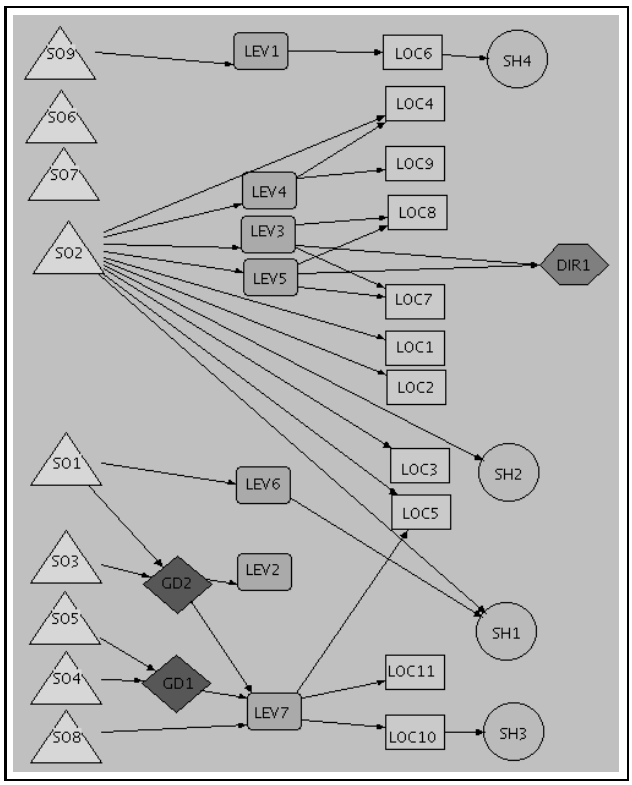


Figure 9: A delegation diagram, showing how the responsibilities are distributed in a design

overall design. Much of the information needed to generate UML diagrams such as sequence diagrams, collaboration diagrams, and class diagrams is already available to Ukase. We also want to use the development of such diagrams to drive the design within Ukase. We already have a prototype web-based tool for drawing sequence diagrams [10]. The standard actions in that tool are to indicate that a message is sent from one object to another — in many cases, this is just specifying that delegation is to take place.

We have also been developing a new kind of diagram, the *delegation diagram*. This shows the delegation relationship between *responsibilities* (as opposed to relationships between classes). This diagram provides a view of the design that shows the static aspects of a class diagram and the dynamic aspects of a sequence diagram.

Figure 9 shows what a delegation diagram might look like for one design resulting from using DSR to build a Sokoban system. It shows classes as columns of as different shapes (and colours), with each instance of a shape corresponding to a responsibility in the class. The triangles on the left hand side are the system responsibilities (actually, responsibilities of the system object), **GD** identify responsibilities for the **GameDescription** class, **LEV** for the **Level** class, **LOC** for the **Location** class, **SH** for the **Shelf** class, and **DIR** for the **Direction** class. The labels used to identify the responsibilities are the same as those used in Ukase (see also Figures 4 and 5). Edges between two shapes indicate that one responsibility is being (possibly partially) delegated to another.

One can determine how a particular system responsibility is met by looking at the subgraph associated with that system responsibility. For example, **SO8** is the system responsibility "Set the current Level to its initial configuration". In this particular design, this responsibility is met by asking a **Level** object to be created (**LEV7**). The **Level** class meets this responsibility by asking (several) **Location** objects to create themselves (**LOC10**), and also to report whether they contain the worker (**LOC11**) or a shelf (**LOC5**). In order for a

Location object to create itself, it has to ask a **Shelf** object to create itself (**SH3**).

One can also determine how a class responsibility is met. For example, the responsibility of the **Level** class "Change location of worker to specified location" (**LEV4**) is partially met by the **Location** class responsibilities "Record that now empty" (**LOC4**) and "Record that now contains worker" (**LOC9**) (see also Figure 7). There are of course other ways that these responsibilities could be met, which would result in different delegation diagrams.

The simplest form of support Ukase could provide for delegation diagrams is to generate them for a given design. It could also allow the designer to use the diagram to navigate the design. Further support would include allowing the designer to use the design itself to specify the distribution of responsibilities as a form of visual programming.

Other additions we want to consider include better support for iterative and incremental design, and version control. We also want to add a reporting facility for such things as impact analysis ("If this requirement is changed, what parts of the design will be affected") and quality control ("Are all the requirements implemented?", "Are there parts of the design that don't come from the requirements?").

As with other design support tools, Ukase records enough of the design that it could automatically generate parts of the implementation. This is considered a long-term goal.

The current Ukase implementation is proof-of-concept. Aspects of it are somewhat crude (witness the "ascii arrows"). There are also a number of places where hyperlinks could be used to better advantage. It has also outgrown the implementation technology, making the additions proposed above difficult to do. There is a reimplementation effort currently underway using JavaServer Pages[17]. The current version of Ukase is being used for the design of the new version. Once the new version is ready, we will carry out usability studies.

5 Related Work

There are a number of products available that support requirements traceability. We believe that RequisitePro [16] is representative of many of them. It also maintains the requirements as use cases, but it uses the RUP-style use cases rather than EUCs. It allows the requirements engineer to identify relationships between documents that are used to develop the requirements (such as the vision document) and the use cases. It also can be integrated with other tools in the Rational family such as Rational Rose [15], and relationships between the requirements and the design can be established. It can provide reports on the status of the relationships, allowing quality questions to be answered.

In contrast to Ukase, establishment of the relationships is done as an explicit step. That is, a decision is made about (for example) a design choice, and the resulting relationship is then recorded using RequisitePro separately. The fact that it is a separate step means that it is possible for the RequisitePro database to become inconsistent with the true state of the design process. With Ukase, the relationships are a direct byproduct of the design process, and so there is never any inconsistency.

6 Conclusions

In this paper, we have presented Ukase, a web-based tool that supports traceability between the requirements, maintained as essential use cases, and an object-oriented design. Unlike other requirements engineering and CASE tools, Ukase maintains traceability between requirements and design automatically as part of its support for the Distribution of System Responsibilities design technique. Our experience with it so far suggests that it does provide effective support for DSR and for traceability between requirements and design.

We have further development work planned for Ukase. As well as a reimplementation, we expect to add visualisation support. We also plan to do usability studies. Once these are complete, we will use Ukase to study the effectiveness of DSR.

References

[1] K. Beck and W. 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] R. Biddle, J. Noble, and E. Tempero. Essential use cases and responsibility in object-oriented development. In *Twenty-Fifth Australasian Computer Science Conference*, Jan. 2002.

[3] R. Biddle, J. Noble, and E. Tempero. Sokoban: a system object case study. In J. Noble and J. Potter, editors, *Proceedings of the 40th International Conference on Technology*, volume 10, Sydney, Australia, 2002.

[4] R. Biddle, J. Noble, and E. Tempero. Supporting reusable use cases. In C. Gacek, editor, *Seventh International Conference on Software Reuse (2002)*, pages 210–226, Austin, TX, USA., Apr. 2002. Springer. LNCS 2319.

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

[6] N. C. for Supercomputing Applications (NCSA). The common gateway interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[7] O. C. Gotel and A. C. Finkelstein. An analysis of the requirements traceability problem. In *First International Conference on Requirements Engineering*, pages 94–101, Colorado Springs, Colorado, Apr. 1994. IEEE Computer Society Press.

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

[9] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[10] R. Khaled, D. MacKay, R. Biddle, J. Noble, and E. Tempero. A lightweight web-based case tool for sequence diagrams. In *SIGCHI-NZ Symposium On Computer-Human Interaction*, July 2002.

[11] M. Lindvall and K. Sandahl. Practical implications of traceability. *Software—Practice and Experience*, 26(10):1161–1180, Oct. 1996.

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

[13] J. Nielsen. *Usability Engineering*. Academic Press, New York, 1992.

[14] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, 1998.

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

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

[17] Sun Microsystems, Inc. Javaserver pages™ white paper. http://java.sun.com/products/jsp/whitepaper.html, 2001.

[18] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

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