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

Ewan Tempero,James Noble, Robert Biddle

Technical Report CS-TR-02/27
September 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

Ewan Tempero,James Noble, Robert Biddle

Technical Report CS-TR-02/27
September 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 use a design technique that, with appropriate tool support, can significantly reduce that cost. It represents the requirements using a particular kind of use case, and develops an object-oriented design. We present Ukase, a web-based tool that provides such support.

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

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.

Tool Support for Traceability between Requirements and Design

Ewan Tempero
Department of Computer Science
University of Auckland
New Zealand
ewan@cs.auckland.ac.nz

James Noble, Robert Biddle
School of Mathematical and Computing Sciences
Victoria University of Wellington
New Zealand
robert@mcs.vuw.ac.nz, kjx@mcs.vuw.ac.nz

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 use a design technique that, with appropriate tool support, can significantly reduce that cost. It represents the requirements using a particular kind of use case, and develops an object-oriented design. We present Ukase, a web-based tool that provides such support.

Keywords*: web-based tool support, traceability, use cases, 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. Maintaining these relationships is expensive without good tool support. We believe there is a particular problem in maintaining traceability between requirements and design. 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 maintaining traceability requires the designer to manually record those relationships, then it significantly increases the cost of the design process. Furthermore, it introduces the likelihood of inconsistency between the actual relationships and those that have been recorded. This reduces the benefits of maintaining traceability in the first place. The question then is, how to maintaining traceability without increasing the cost of the development process. Ukase demonstrates that this cost can be reduced almost to zero.

The most effective use of Ukase requires a subtle, but significant, change in the design process. The basic thinking processes are the same, but they are organised in a particular way. The emphasis is on distributing *responsibilities*, and so we call this *Distribution of System Responsibilities* (DSR). DSR represents the requirements using a particular kind of use case, and develops an object-oriented design. Ukase supports DSR by managing the distribution of responsibilities, and in doing so, automatically records traceability relationships both between use cases and classes, and between different classes.

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.

[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.)

2. Background

2.1. 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 might include: "Why is this requirement here?", "Do the requirements satisfy the needs of the client?", "What alternative decisions were considered?", "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 requirement, in both a forwards and backwards direction" [8]. 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* [18]. 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 [12]. 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.

Traceability is an important consideration in evaluation of a design, as it can be used by evaluators to check whether a given design in fact meets the specified requirements. Specific reading techniques have been developed and validated for evaluation using this approach, and involve both vertical and horizontal traceability [24].

2.2. Use Cases

Ukase represents the functional requirements as a particular form of use cases, based on *essential use cases* (EUCs) [6, 7]. Jacobson et al. define a use case in their 1992 book as "a behaviorally related sequence of transactions in a dialogue with the system"[10]. 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"[9]. 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"[6]. 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 [27].

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 *user intentions* and *system 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. The important observation is that the set of system responsibilities from all essential use cases represent the requirements of the system.

The essential use cases style is better suited to our purposes because it makes it easy to determine what the functionality of the system is, and the abstract nature of the steps places fewer restrictions on the design process. There is, however, no reason why DSR could not be applied with other styles of use case.

2.3. Distribution of System Responsibilities

Distribution of System Responsibilities (DSR) is a technique to develop 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 fulfil. 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]. DSR essentially organises the CRC process to make the decisions made about which responsibilities go in which class explicit — these decisions are the traceability links that Ukase records.

The starting point for DSR is to gather all of the system responsibilities identified in the use cases 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 making decisions about how to do the distribution. A consequence of following 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.

DSR is a conceptual model, or frame of reference for thinking about design, or more particularly, the delegation relationships between requirements and design. DSR does not constrain the order of the design process, or restrict the kinds of final designs that are produced. This means DSR can accommodate design decisions that depend on the domain model, software architecture, the use of design patterns, and/or frameworks. By viewing the design process as the distribution of responsibilities latent in the requirements of the system, DSR ensures that the information required to capture traceability requirements is explicitly generated as a natural side effect of the design process. Further details are available elsewhere [2, 3].

Ukase supports DSR by first supporting the capture of requirements as use cases, and then by supporting the process of deciding how to distribute the system responsibilities identified in the use cases 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.

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 [25] CGI script [15], with the information stored in a MySQL database [14]. It is highly accessible (anywhere on the Internet), and may be used as groupware

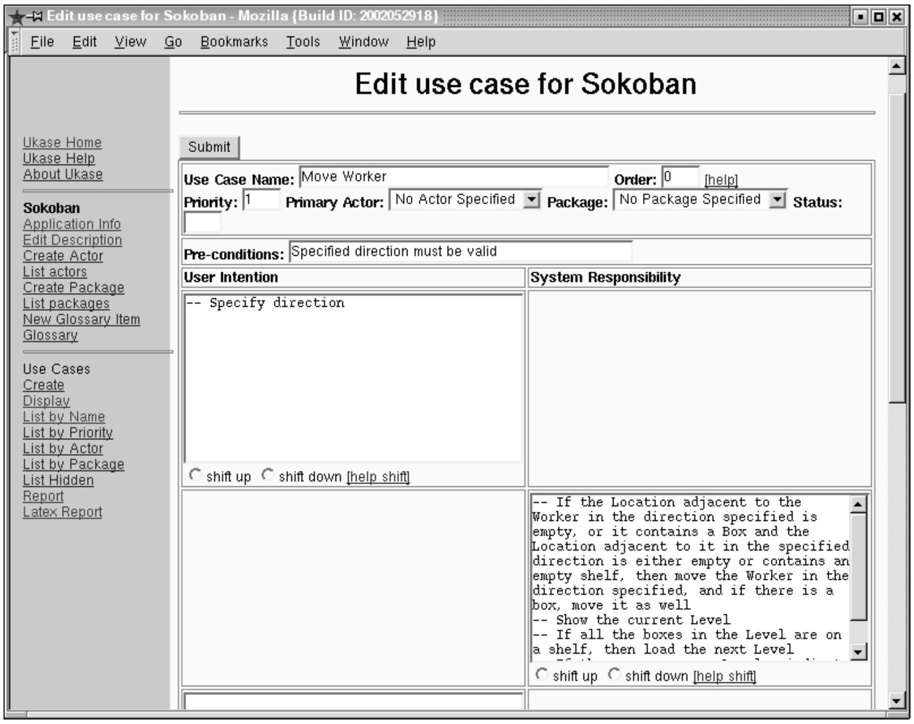

Figure 2. Entering the details of a use case in Ukase.

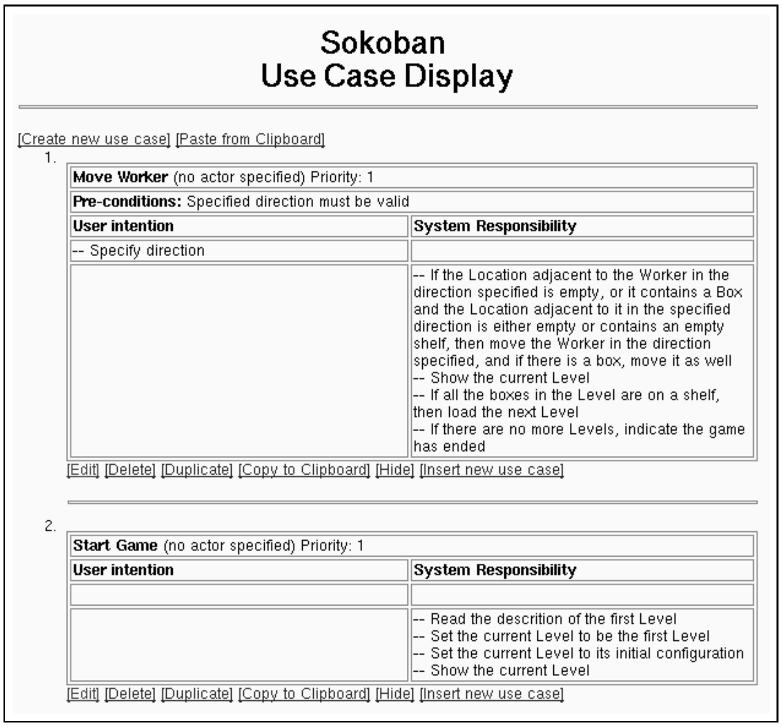

Figure 3. Some of the use cases for Sokoban.

(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.

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].

3.1. Supporting Use Cases

Ukase provides tool support for DSR, and so a significant part of its functionality is management of use cases. Some 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.

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 system 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) 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.

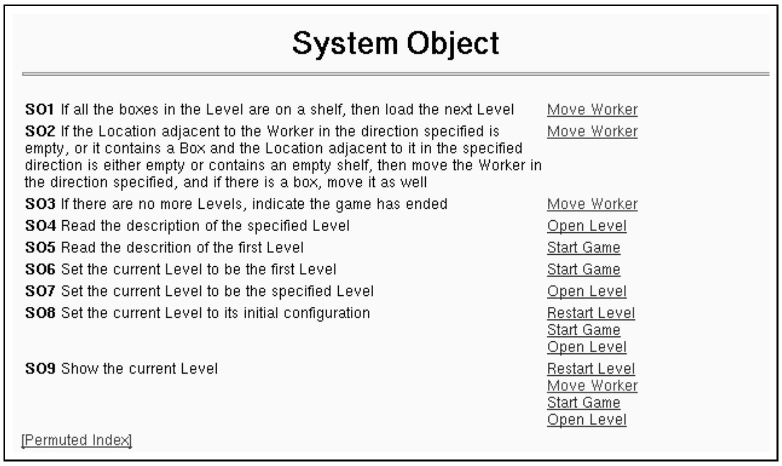

Figure 4. The System object for Sokoban

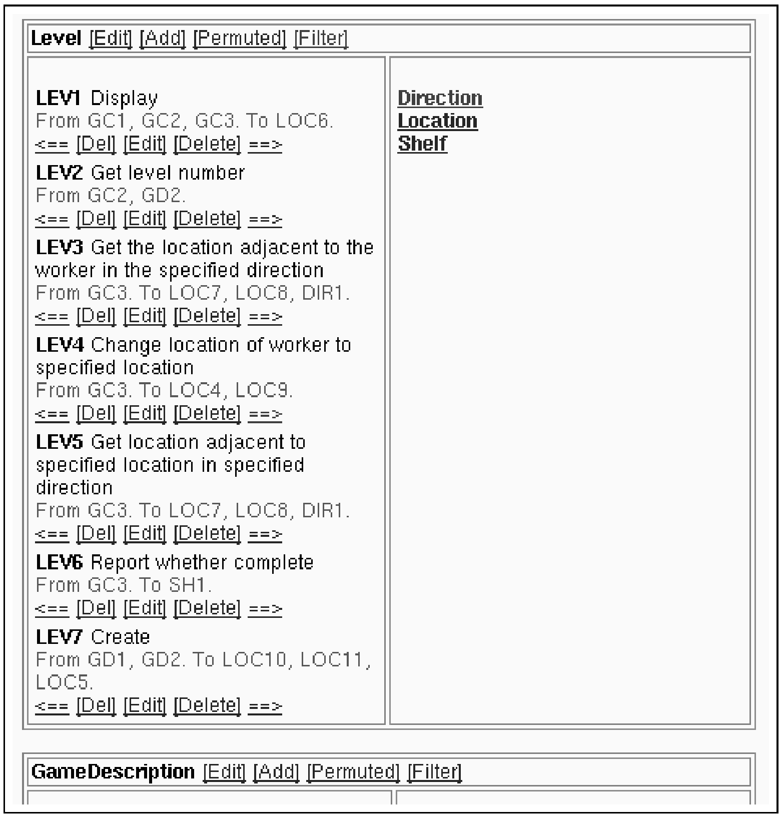

Figure 5. The Level class in a design for Sokoban.

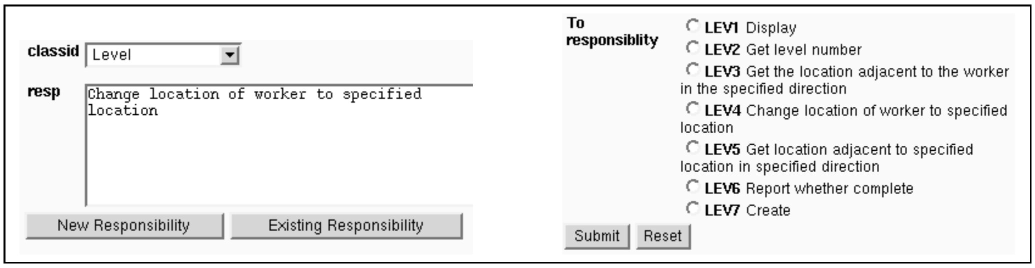

Figure 6. Two forms for delegating responsibilities to the Level class.

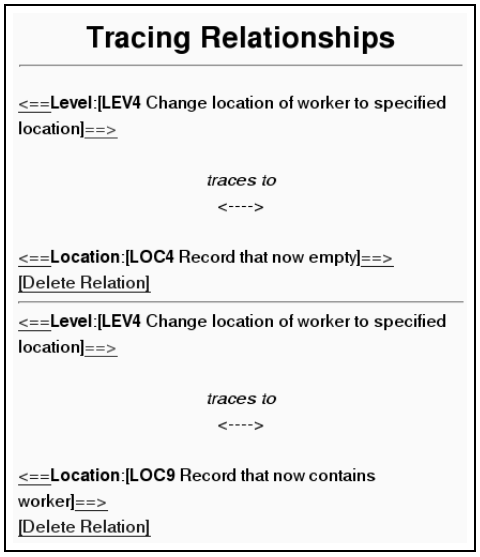

Figure 7. The responsibilities that LEV4 delegates to.

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 9 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 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 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 [16] 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.

We believe that Ukase could also prove useful in design evaluation using design reading techniques that have already been developed [24]. Ukase would make these techniques easier to apply by having many of the critical elements already identified and linked together. The web-based nature of the tool also supports this by allowing evaluation using access available anywhere.

The main addition we are making 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 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 [11]. 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 8 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

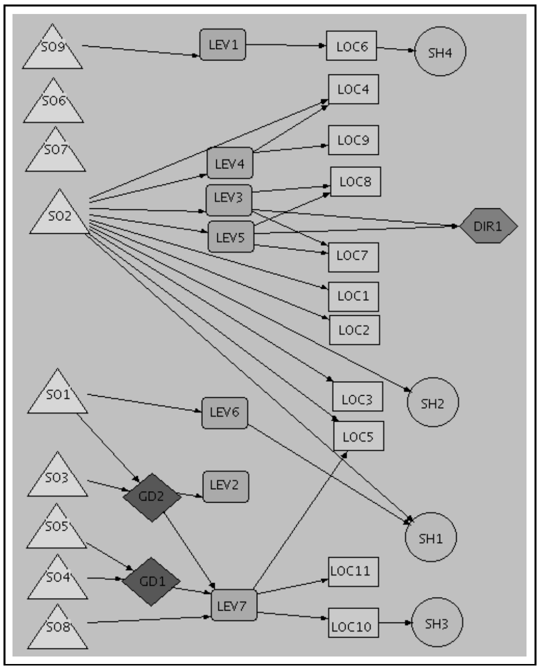

Figure 8. A delegation diagram, showing how the responsibilities are distributed in a design

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 diagram 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 improve the 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?”). While Ukase currently allows these questions to be answered in principle, we would prefer directly support.

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[23]. The cur-

Read the descrition of the **first**	Level
Set the current Level to be the **first**	Level
If there are no more Levels, indicate **game** the	has ended
If there are no more Levels, **indicate**	the game has ended
Set the current Level to its **initial**	configuration
Set the current Level to be the **Level** specified	
Read the descrition of the first **Level**	
Set the current **Level**	to be the specified Level
Set the current **Level**	to be the first Level
If all the boxes in the Level are on a **Level** shelf, then load the next	
Set the current Level to be the first **Level**	
Read the description of the specified **Level**	
Set the current **Level**	to its initial configuration
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
If all the boxes in the Level are on a **load** shelf, then	the next Level
If the Location adjacent to the Worker **Location** in the direction specified is empty, or it contains a Box and the	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
If the **Location**	adjacent to the Worker in the direction specified is empty, or it contains a Box

Figure 9. Part of the permuted index for the system responsibilities for Sokoban

rent 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 has been much work on traceability, particularly in the requirements engineering community. A number of traceability models have been proposed emphasising different uses of traceability, typically with different sets of objects that the models can describe, and different types of links between the objects. This work is well characterised by the work by Gotel and Finkelstein, who discuss the issues involved in the "requirements traceability problem" [8], and that of Ramesh and Jarke, who summarise the important variations in their work on a requirements traceability reference model [20].

Work on requirements traceability has primarily been concerned with capturing rationale for decisions made about requirements. For example, REMAP is a conceptual model for capturing the deliberations involved on alternative requirements decisions [19], Gotel and Finkelstein determine that traceability of artifacts that are used to develop requirements [8] (so called "pre-RS" traceability) is as important as traceability from requirements to the design and implementation artifacts ("post-RS" traceability), and Ramesh and Jarke's reference model is largely motivated by similar concerns. Many of the issues raised in these discussions do apply to the relationships between requirements and design, but there has been little discussion of the issues that apply specifically to these relationships.

Lindvall and Sandahl have been involved in a long-term industry-scale commercial project studying the use of, benefits of, and issues with post-RS traceability [12, 13]. Their earlier work concludes that "traceability-focused development will increase documentation clarity and focusing, thus also providing benefits to earlier life-cycle phases". Their later work finds that, at least in the project they are studying, that developers tend not to use documented traces, preferring to interview the relevant experts. We speculate that this may be due to the fact that documentation is expensive to produce, and difficult to keep up to date. Their study is in an environment where the requirements are recorded as use-cases, and so is very relevant to our work.

There are a number of products available that support post-RS traceability. We believe that RequisitePro [22] is representative of many of them. It also maintains the requirements as use cases, but it uses the RUP-style use cases. 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 [21], and relationships between the requirements and the design can be manually 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 separate 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.

We are not the first to recommend focusing on responsibilities for design. As mentioned above, responsibilities play a key role in CRC [1], and Wirfs-Brock et al argue that basing design on responsibilities improve encapsulation [26]. What we have done with Ukase is to provide explicit support for managing responsibilities.

As mentioned in the previous section, there is a clear relationship between the responsibility "graph" created by DSR, and UML sequence diagrams [17]. We speculate that using a "responsibility mindset" when creating sequence diagrams may provide the same benefits we have seen with Ukase.

6. Conclusions

In this paper, we have presented Ukase, a web-based tool that supports traceability between the requirements, maintained as 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 Ukase so far suggests that it does provide effective support for DSR and for traceability between requirements and design. It demonstrates that traceability between requirements and design can be achieved for almost no cost.

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

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

[8] 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.

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

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

[11] 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.

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

[13] M. Lindvall and K. Sandahl. Traceability aspects of impact analysis in object-oriented systems. *Journal of Software Maintenance: Research and Practice*, 10:37–57, 1998.

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

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

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

[17] Object Management Group. Unified Modeling Language (UML) 1.4 specification, Sept. 2001.

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

[19] B. Ramesh and V. Dhar. Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, June 1992.

[20] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, Jan. 2001.

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

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

[23] Sun Microsystems, Inc. Javaserver pages$^{\mathrm{TM}}$ white paper. http://java.sun.com/products/jsp/whitepaper.html, 2001.

[24] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–56. ACM Press, 1999.

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

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

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