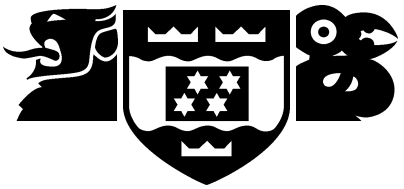
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

Lightweight Web-based Tools for Usage-Centered and Object-Oriented Design

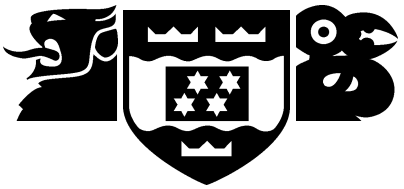
Robert Biddle, James Noble, and Ewan Tempero

Technical Report CS-TR-02/13
July 2002

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

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

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



School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

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

Lightweight Web-based Tools for Usage-Centered and Object-Oriented Design

Robert Biddle, James Noble, and Ewan Tempero

Technical Report CS-TR-02/13
July 2002

Abstract

Every kind of development needs appropriate support. Paper, pencil, and index cards are extremely fine for small examples, but will not scale up to multiple teams of distributed developers, while large-scale CASE tool suites are a rational choice for mission-critical development but require too much investment to be applicable to Agile projects. Lightweight, web-based tools can provide cheap and cheerful support for small to medium sized agile developments, being better at recording and sharing information than paper or generic software tools, but without the overheads and strictures of full-blown CASE systems. We present a range of lightweight, web-based tools for all phases of the software lifecycle, both that we have developed, and that are available over the Internet.

Publishing Information

A version of this paper is to be presented at forUSE 2002, The First International Conference on Usage-Centered Design, Performance-Centered Design, and Task-Oriented Design - 25-28 August 2002, Portsmouth, New Hampshire, USA.

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.

Lightweight Web-based Tools for Usage-Centered and Object-Oriented Design

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

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

July 19, 2002

Abstract

Every kind of development needs appropriate support. Paper, pencil, and index cards are extremely fine for small examples, but will not scale up to multiple teams of distributed developers, while large-scale CASE tool suites are a rational choice for mission-critical development but require too much investment to be applicable to Agile projects. Lightweight, web-based tools can provide cheap and cheerful support for small to medium sized agile developments, being better at recording and sharing information than paper or generic software tools, but without the overheads and strictures of full-blown CASE systems. We present a range of lightweight, web-based tools for all phases of the software lifecycle, both that we have developed, and that are available over the Internet.

1 Introduction

> *There is nothing new under the sun.*
> Ecclesiastes 1:9.

For better or for worse, software engineers never seem to be able to resist the temptation to "go meta" and to attempt to apply the spoor of their practice — software — to alleviate their own struggles in the mire of its development. Now, no-one would want to be the cobbler's child without any shoes, and software has indeed been very profitably applied as a means of its own production and distribution since the first assemblers, not

to mention programming languages, compilers, debuggers, linkers, window painters, task builders, Yaks, Bisons, Superzaps, and all the other well-polished power tools programmers love to keep in their sheds and show off to visiting relatives.

While the benefits of these low-level software tools are well accepted, the benefits of higher level Computer Aided Software Engineering (CASE) tools that are not directly related to program code are rather more dubious. These tools are aimed at the earlier stages of the software lifecycle: typically supporting notations such as data-flow diagrams and structure charts — or rather, their 1990s equivalents such as class diagrams and sequence charts.

Problems With CASE Tools

Unfortunately, experience with CASE tools has been nowhere near as positive as with more basic software tools. These CASE tools seem to have a number of problems. The largest is probably the distinction software design tools seem to promote between the "design" and the "software" — that is, between the map and the territory. The ultimate aim of software development is just that, software, or to put it more crudely: a program. Pictures, diagrams, or designs recorded within CASE tools are useful only inasmuch as they contribute value to the final program. Program code itself is a design (that is, an abstract description) for a piece of software: a compiler constructs the physical representation of that design, and then a CD duplicating plant actually manufactures copies of that physical representation.

This means that CASE tools can win only half the battle: first because they do not provide enough detail to construct all the software, so developers still have to work at both the CASE tool and programming language level. More importantly, even those round-trip tools that can replace *programming languages* cannot replace *programming*. Rather, programming simply happens within the confines of the CASE tool, in just the same way that FORTRAN or AUTOCODE did not replace programming: we now programming in these languages or their successors, rather than writing binary code by hand.

Early in the software development lifecycle, however, we may not wish to start by programming, managing a large amount of information in detail and depth, whether directly in a programming language or indirectly in a CASE tool. In the early stages of design, this amount of detail is simply not required. A designer may wish to experiment with design ideas, or quickly produce a sketch for inspection by users or customers. For these tasks, more flexible techniques that capture less information but quicker to use are much more appropriate.

Finally, CASE tools generally inflexible and difficult to use, because the are designed to *enforce* a particular software development process. CASE tools generally provide a form of syntax-directed editing, ensuring that "incorrect" diagrams cannot be created, via local syntax checks on individual diagrams (so, for example, a class cannot inherit from itself) and global consistency checks across entire models (so that every object in a sequence diagram must belong to a class defined in a class diagram). Syntax directed editing has failed for programs: and has arguably failed for CASE tools too. The problem is that when editing programs (or documents or texts more generally) people do not always maintain either local or global correctness: imagine editing a

memo on a word processor that automatically deleted any word that was grammatically incorrect or misspelt.

The Baby and the Bathwater

As a reaction to the problems with CASE tools, Agile methodologies advocate the use of pencil and paper techniques such as index cards or whiteboards (Highsmith 2002). These "real world" techniques are typically fast and easy to use, completely unconstrained in notation, and multi-user. Paper diagrams can be easily copied for distribution, or filed for future reference, and can be read by everyone.

Of course, pencil and paper techniques have several disadvantages. Large scale editing operations (such as moving part of a diagram while keeping its connections intact) are difficult to perform. Paper diagrams are difficult to place under automated revision control, and associated process information is difficult to capture. This can create serious practical problems dealing with even medium-sized programs: even searching a few hundred cards' worth of objects or use cases can be very difficult. The information in pencil and paper diagrams or CRC cards cannot be extracted easily for input to other CASE tools. While Agile development advocates have argued in favour of not writing anything on index cards, and erasing whiteboards as soon as the design session has finished (and before the "real work" of coding begins) these strictures actually illustrate the difficulty of maintaining important information on such ephemeral media — how many whiteboards have "Do Not Erase" notices on them?

Lightweight Tools

With the rise of the web and scripting languages, it is now possible to find alternatives to both large CASE tools on one hand, and pencil-and-paper techniques on the other. Technology has advanced so that simple, useful, *lightweight* tools are directly available freely, or can be built as cheaply as necessary using materials that are ready-to-hand. Essentially, a lightweight tool is small, requires little effort to learn to use effectively, imposes few process restrictions on how it can be used, and can be implemented simply.

The key advantage of building and using lightweight tools is that they can have a very high usability, in the sense of a good fit to the actual usage that will be made of them. In our case, we have built several tools to support the notations and processes that we actually use — such as Essential Use Cases for Usage-Centered Design, or sequence diagrams for OO design. These tools directly support the mundane tasks that are part of software development, such as tracking work completed, reliable recording of artefacts, providing an audit trail, and providing publication-quality output to impress managers. Furthermore, being web-based, lightweight tools can naturally provide good support for multi-user group work and distribution. If a lightweight tool offers minimal functionality — supporting just those tasks that are required for actual development — the tool can be smaller as a whole and thus easier to learn. Finally, lightweight tools can be infinitely tailorable to suit whatever notation or process you are using, especially if you are building them yourself.

Lightweight tools have another benefit —- often as a side effect of their implementation technology, or simply because they are often built "in house" and so not finished

to commercial standards. Lightweight tools tend to be ugly. This ugliness can be especially pronounced in web applications. Note that one of the gifts of the web is the general acceptance of low complexity interfaces for simple functionality — although most web applications, even commercial offerings such as Hotmail or Yahoo groups are generally uglier than the equivalent desktop software.

The advantage conferred by ugliness can be summed up in the catchphrase *"messy is good"* (Noble and Biddle 2002). Ugly, unpolished interfaces communicate cheapness to their users. Programmers and designers are less likely to lavish attention on their models and diagrams if they are cheap to produce, and are more likely to change them if they always look unfinished. More importantly, ugliness helps remind programmers that their early models and designs just that, models or sketches, and are not themselves the final output of the development process. Overall, the aim is that developers should need to invest less time and effort, both into the lightweight tools themselves, and also the artifacts and models the tools support.

In the remainder of this paper, we present three lightweight, web-based tools which we use to support Usage-Centred and Object-Oriented Design. Ukase supports Essential Use Cases, Seek supports sequence diagrams, and a WikiWikiWeb server acts as a group-wide shared memory.

2 Ukase: Essential Use Cases

Our first example of a lightweight web-based tool is Ukase[1], a tool we have developed to support essential use case models (Constantine and Lockwood 1998) (See Figure 1). The original purpose of Ukase was just to make the recording of use case models easier, but we quickly found it was useful for the development of the models themselves (Biddle, Noble and Tempero 2002).

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 (Wall, Christiansen and Orwant 2000) CGI script (for Supercomputing Applications (NCSA) n.d.), with the information stored in a MySQL database (Mys n.d.). It is highly accessible (anywhere on the Internet), and may be used as group-ware (multiple people can access a model simultaneously).

As mentioned above, Ukase provides support for recording essential use cases. It 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 CGI form laid out the way that a use case would look in a report (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 1, and once development is complete, the resulting model

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

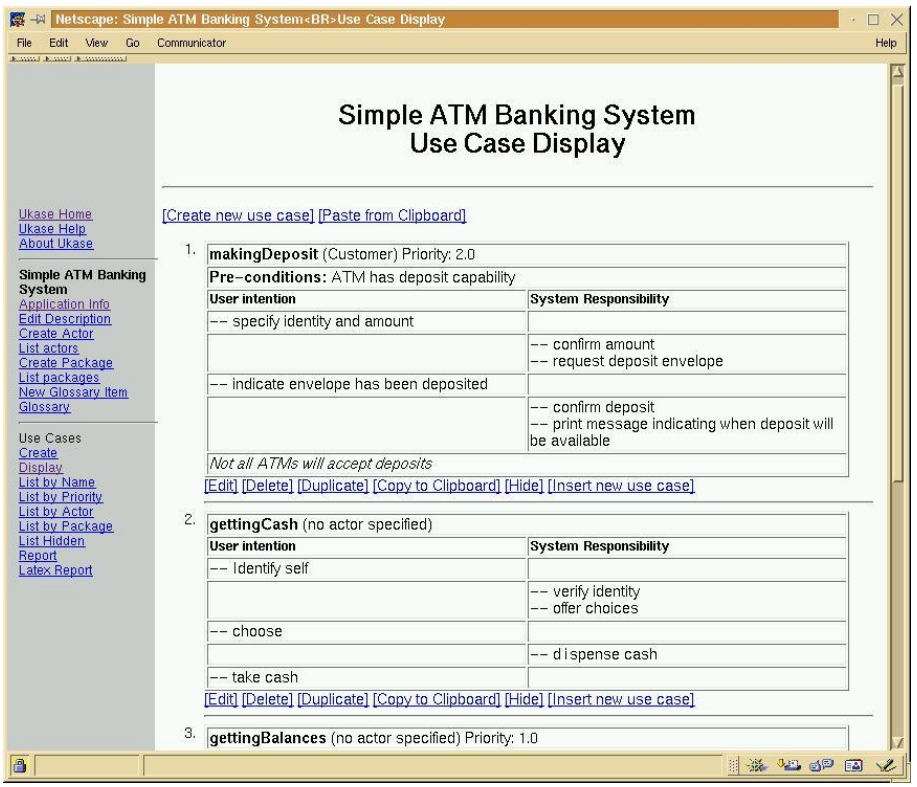


Figure 1: Ukase is a web-based tool for managing essential use cases.

is available as a report (figure 3). Use cases can also be listed in a variety of ways (figure 4, for example).

Any use case model typically uses many terms particular to the application domain, which can make the intent difficult to understand. Ukase provides a *glossary*, where such terms can be more fully explained. Some of these terms are often concepts that would show up in an *analysis model*, and so Ukase allows such terms to be tagged as "candidate" classes (figure 5).

3 Seek: UML Sequence Diagrams

Our second example of a lightweight tool is Seek (Khaled, McKay, Biddle, Noble and Tempero 2002), a diagram editing tool for sequence diagrams in the Unified Modelling Language (UML) (Group 2000) notation implemented by Rilla Khaled. A sample screen is shown in figure 6. The tool is intended to facilitate the early part of the design process, where sequence diagrams are frequently used to explore and explain object interaction. The tool is web-based to maximise accessibility and use by teams, and uses a simple yet effective image tiling technology to allow manipulation of the

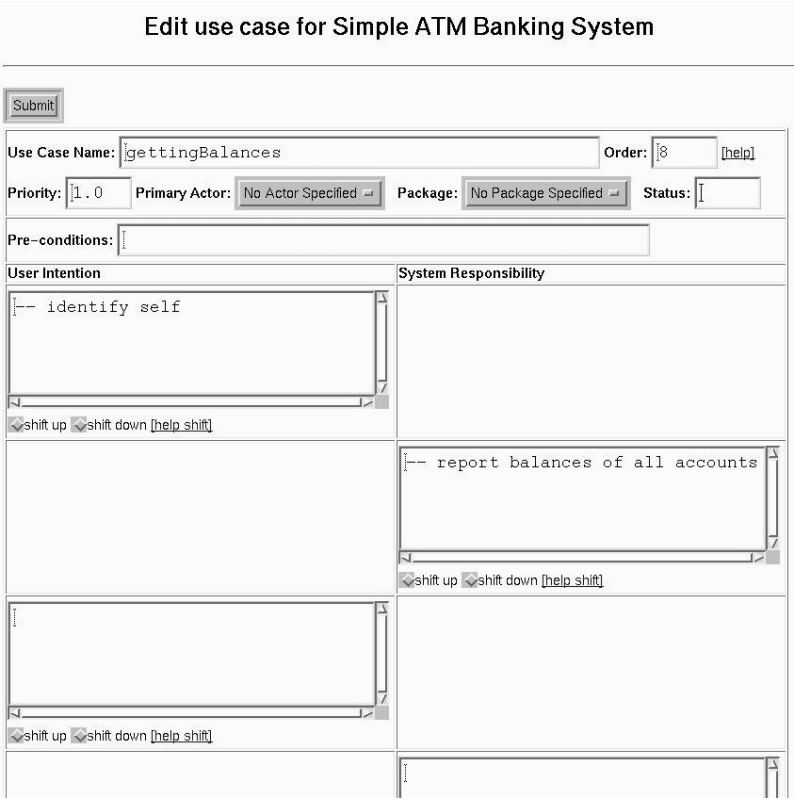


Figure 2: Entering the details of an essential use case in Ukase.

Use Cases

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

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

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

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

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

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

4. **listingAccounts** (Customer)

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

5. **performingSelfTest** (Servicer)

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

Figure 3: Showing all the details of the use case model.

Simple ATM Banking System
Use Cases grouped by Actor

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

Actor Servicer
1. [Edit] – performingSelfTest

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

Figure 4: Listing the use cases grouped by primary actor.

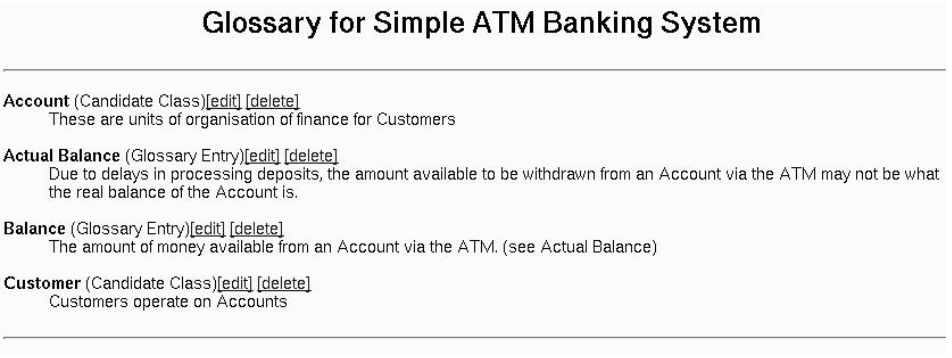


Figure 5: The glossary, showing some of the terms that might be useful in the domain model.

diagrams from any web browser.

The challenge that we faced when designing Seek was that web support for interaction with diagrams is either very limited, or involves applets or plug-ins that were incompatible with ubiquitous web access. We decided to accomplish what we could using the very limited interaction available to any browser.

The user interface technology consists simply of HTML pages and forms, using hyper-links with images. Our implementation uses Java Server Pages (JSP) together with JSP JavaBeans (Sun Microsystems Inc. 2002, Bergsten 2000). The JSP is used to respond to form requests and generate the HTML pages; a MySQL database is used to store the state between displays. This structure is depicted in figure 7.

The tool allows diagrams to be created, edited or deleted, and several diagrams can be grouped together as “portfolios”. Seek also supports simple portfolio management activities. Seek lends itself well to use in teamwork situations as a team may own one or several portfolios, the diagrams are accessible from anywhere, and a simple form of diagram locking is provided.

An unusual aspect of our implementation is how the tool allows effective interaction with the limited support available in common browsers. We represent a sequence diagram as a two dimensional grid of image tiles, as shown in figure 8. Each tile is displayed using one of a set of images that depict each of the possible action states within a sequence diagram. The set of tiles is shown in figure 9. When combined in correct configurations, the small images collectively represent a UML sequence diagram.

The set of images is displayed as a palette at the left on the tool interface. The sequence diagram is created by repeatedly selecting an image by clicking on a palette image, and then clicking in the grid where that image should be displayed. The clicking is in fact activating hyper-links, which cause the state of the diagram to be changed and then redisplayed. The images are very small and cached at the browser, so the redisplay is quick.

The simplicity and responsiveness of the image tiling technology supports the lightweight nature of the tool. A further consequence of drawing diagrams by building them out of a limited number of small “action” images is that the creation of correct

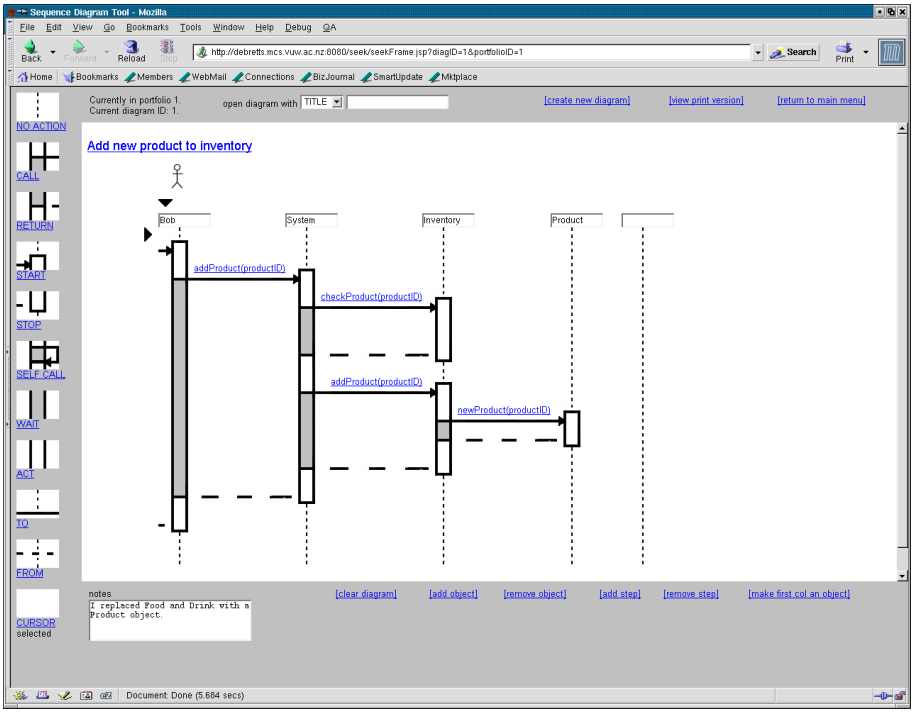


Figure 6: The main screen of Seek, showing a sequence diagram and facilities for editing and manipulation of the diagram. The diagram shows a use case, beginning with an actor at the left, and showing the step-by-step interaction between the objects.

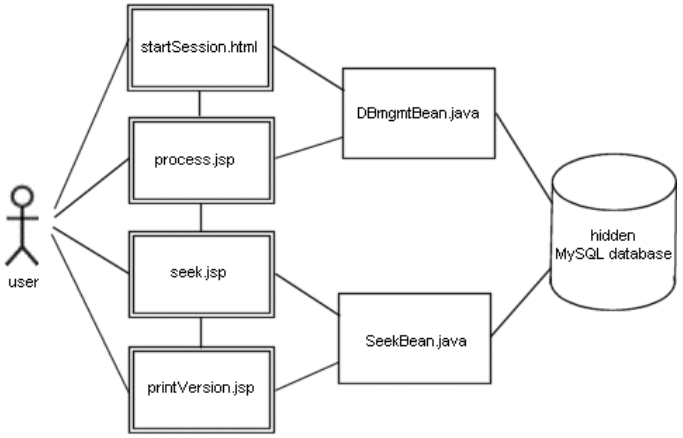


Figure 7: The architecture of Seek.

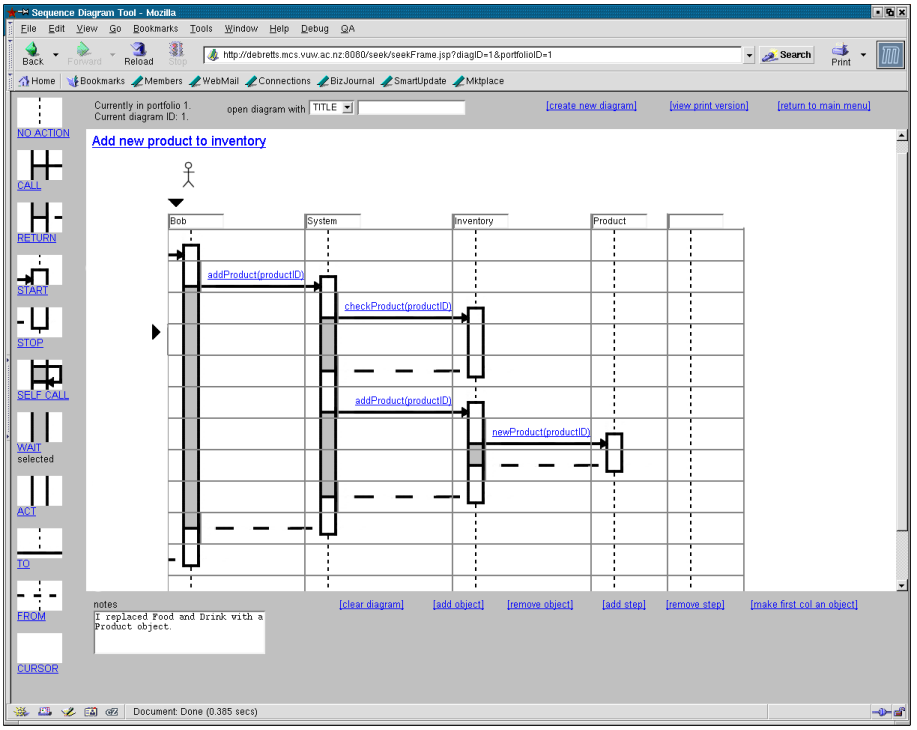


Figure 8: The same screen illustrated in figure 6, but showing the image tiles that form the diagram, and that allow the diagram to be manipulated.

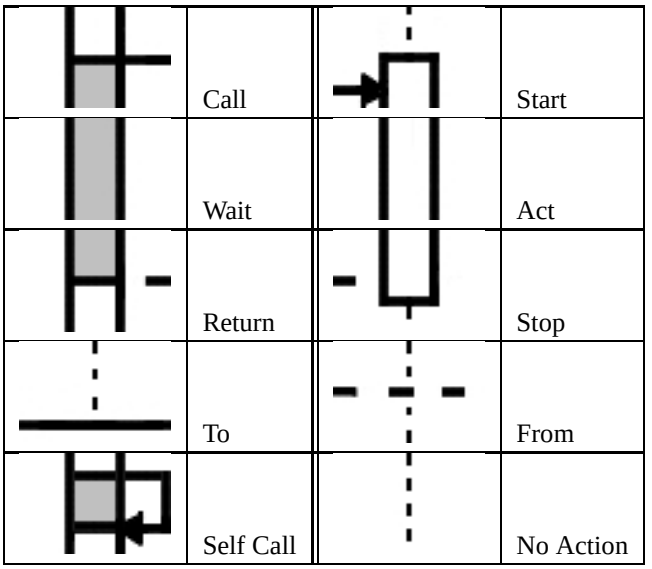


Figure 9: The images used to created a tiled sequence diagram: each image matches a particular state or action in a thread of execution.

diagrams is partially supported, because after a short time of using the tool, it becomes evident which actions should be selected to be used above, below and on either side of other actions, due to how they correctly "fit" together. To further encourage creation of correct diagrams, the tool does simple error checking, and reports errors by pointing with small red arrows at the places where "illegal" diagram configurations exist. Supporting correctness of diagrams may help novice programmers better understand composition, collaboration, and object interaction, however Seek also tolerates incorrect diagrams so that it does not restrict expert developers.

Seek is limited in what it can draw: it cannot represent parallel system calls, and right-to-left system calls. While the tool could easily be modified to handle right-to-left system calls, the intended use for the tool was not to represent complex scenarios but rather standard scenarios, which it can indeed depict. Additionally, it does not specify return types for methods — while it could be easily modified to do so, displaying the return types could lead to a cluttered diagram and specification of return types is not a UML convention that we use.

Diagram editing is done from Seek, which contains the drawing grid for the diagram. To make a change to the drawing grid, which initially contains the dotted deactivated lifelines of 8 objects, the user must first select an action from the sidebar on the left-hand side by clicking on it. In order to apply the tool to a certain grid square, the user must then click in the relevant grid square. Clicking in the drawing grid results in an automatic update and save to the database records for the diagram, and a screen refresh so that the square clicked in will now contain whatever image the action happens to be. In other words, Seek has a concept of a "cursor" that gets updated to

point to the last place where the user intended to apply an action, and then when the diagram refreshes to show a change, the change is made to wherever the cursor was last pointing.

The grid update method changes some grid contents automatically by employing a "smart-tiling" policy — if a start image is already contained above a stop image that the user has just added to the diagram, then as long as there are blank grid spaces between them, the blanks are replaced with act images. The reverse case also causes an automatic update, i.e. if a start image is added directly above a stop image. The same sort of thing is done when a return is added below a call (the space between them is changed to wait images) or if a call is added above a return. Likewise, to lines are drawn between a call image in line with a start image (to the right) and from lines are drawn between a stop image and a return image (to the left).

The diagram features many embedded facilities to support editing. To add and remove steps or objects into the diagram, the user can either just select the appropriate link from the bottom toolbar, for example "add step", "delete object", and so on to add or delete steps or objects from the last position the cursor was stationed at. If they want to change the position of the cursor before adding steps, without carrying out an action, they can select the "move cursor" tool and click where they want the cursor to now point to. From this page, objects can also be given names (by filling in the form boxes at the top of the page), method names can be changed (by clicking on the hyperlinks that say `call()`), diagrams can be given titles, and also have "notes" written about them.

There is also a link to display a non-interactive representation of the diagram, as shown in figure 10. This allows the user to print or capture a diagram suitable for inclusion in other documents.

4 WikiWikiWeb: Lightweight Hypertext

Our last example of a lightweight tool is the WikiWikiWeb, an editable hypertext system, originally created by Ward Cunningham and now with many imitators worldwide (Leuf and Cunningham 2001). The original WikiWikiWeb is also implemented using Perl via a CGI Gateway, although other implementations are based on every kind of web technology imaginable. Most WikiWebs do not even use a SQL database, but rather store information via Perl's single file database. We have used WikiWebs as unstructured content repositories, acting as a shared group memory for software development projects.

Figure 11 illustrates a page from a WikiWeb. Essentially, a WikiWeb page is a standard web page, accessed over the Internet or an intranet like any other web page — although its web design is typically more spartan than most. While Wiki pages can contain some formatting — text may be monospaced, bold, or italic, pages can use bulleted and numbered lists, and include pictures and external URLs (as monospace text beginning http:), many of the more advanced formatting features of the web are simply not supported. WikiWebs also make curious use of BiCapitalised words within text (see `ClassDiagram` and `SmsSupport`) from figure 11): these link to other pages on the same Wiki server, and so have been highlighted as hyperlinks by the web

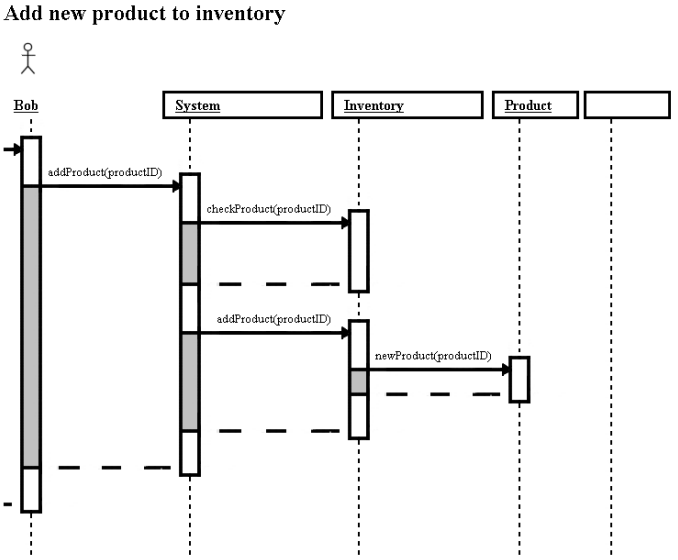


Figure 10: An image generated for printing based on the diagram created as shown in the earlier figures.

browser.

The key feature that separates WikiWeb pages from standard web pages is that there is an `EditText` link at the bottom of every page. Following this link leads to an editing page (shown in figure 12) that can be used to update or replace the content of that page. The editing page is the most lightweight (and most ugly) part of the whole WikiWikiWeb interface: simply a text box and a `Save` button.

This text box is the key to the WikiWikiWeb, that makes is possible for anyone to edit pages quickly and easily. The contexts of the text box are treated as a very rudimentary markup language that is used to produce the web pages. This markup language is carefully designed to support only the most crucial features and thus be simple to learn and use. For example, figure 13 lists the core of the Wiki "markup language" is.

This is clearly a much smaller language than HTML, and also much efficient (bullet lists require two characters per item, for example, compared with an overhead of ten for each list plus for four per item in HTML). Of course, many HTML features are simply not supported in Wiki — tables, justification, background colours, image positions, free choice of typeface font and colour — but these features are not essential to the basic hypertext support. More importantly, because Wiki does not support these features they are not expected (nor missed) by habitual users: removing options about the *form* of the pages leads to an increased focus on the *content*.

The design of links between pages in Wiki is a final example of a brilliant lightweight minimalist design. Wiki pages are not organised into any kind of hierarchical structure, rather a Wiki server provides a flat namespace of uniquely named pages. All page

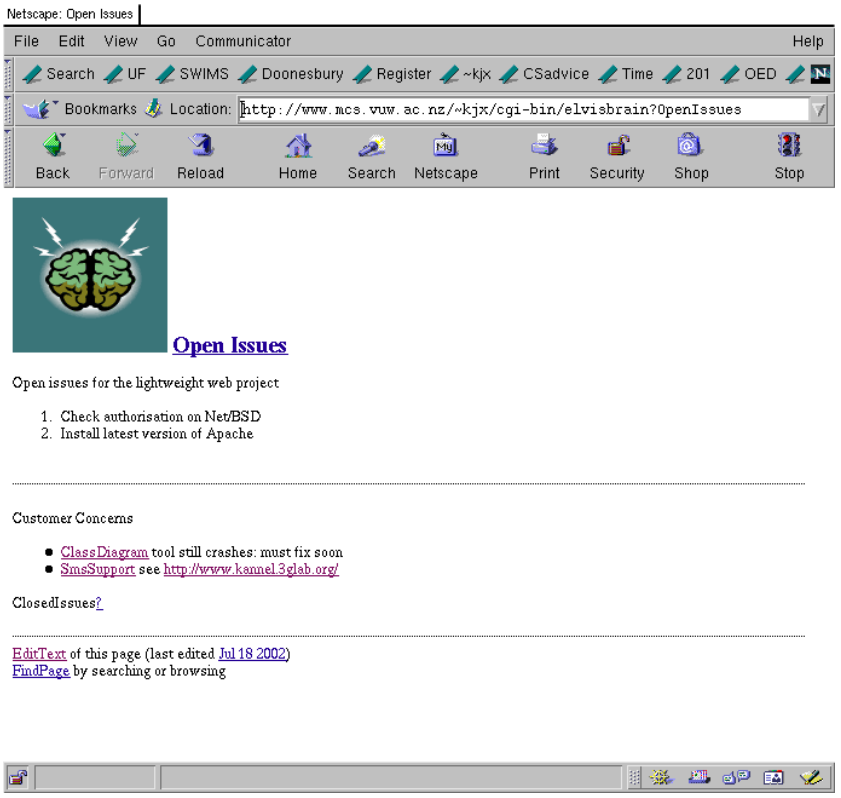


Figure 11: A WikiWikiWeb page.

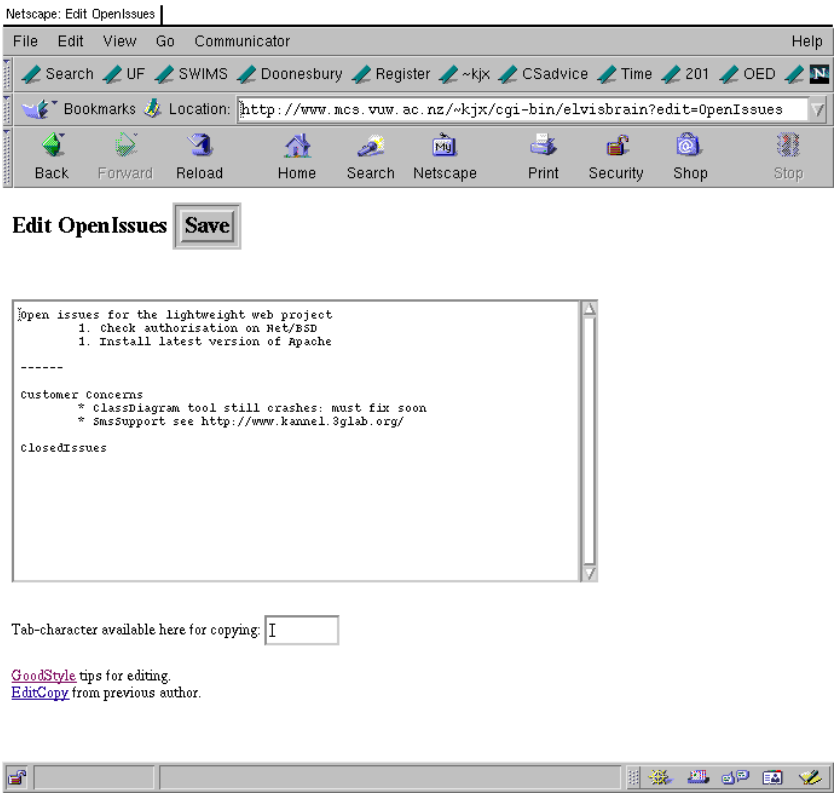


Figure 12: Editing a WikiWikiWeb page.

Blank link	Paragraph Break
Line containing `-----`	Horizontal rule across web page
Line begins with `<tab>*`	Bulleted list item
Line begins with `<tab>1.`	Numbered list item
`'foo'`	foo in italic
`''foo''`	foo in bold
URL `http://...`	if the URL is to a web page it is typeset as a literal and forms a link if the URL is to an image, the image is inserted inline.
`BiCapsPageTitle`	Link to another page titled `BiCapsPageTitle`

Figure 13: WikiWikiWeb Markup Language

names must be BiCapitalised, and the text markup language recognises all BiCapitalised words as links to pages and formats them as such. If the page does not exist, a link is formatted ending in a question mark: clicking on the question mark will create a new blank page ready for editing (see Figures 14 and 15).

Figure 14: Page links (a) before and (b) after creating the page.

5 Conclusion

In this paper we have presented several lightweight, web-based tools that we use to support agile usage-centered object-oriented design practices.

These tools are tailorable (or more to the point, can easily be constructed to suit) the context and environment of a particular development and so support the actual development practices that we use. Ukase, for example, supports the exact form of Essential Use Cases we use in all our developments, rather than the more clumsy Rational Unified Process use cases. Similarly, Seek supports just those parts of UML sequence diagrams that we have found useful. The popular lightweight WikiWikiWeb is as effective as heavyweight knowledge management tools for small to medium sized agile development projects. And, by following the maxim "Messy is Good", these tools remind us that the ultimate point of usage-centered object-oriented buzzword-compliant modelling is to ensure that there will good software at the end of it.

References

Bergsten, H.: 2000, *JavaServer Pages*, O'Reilly & Associates.

Biddle, R., Noble, J. and Tempero, E.: 2002, Supporting reusable use cases, *Seventh International Conference on Software Reuse (2002)*, pp. 210–226.

Constantine, L. L. and Lockwood, L. A. D.: 1998, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, Addison-Wesley.

for Supercomputing Applications (NCSA), N. C.: n.d., The common gateway interface, http://hoohoo.ncsa.uiuc.edu/cgi/.

Group, O. M.: 2000, Unified modeling language (UML) 1.3 specification.

Highsmith, J.: 2002, *Agile Software Development Ecosystems*, Addison-Wesley.

Khaled, R., McKay, D., Biddle, R., Noble, J. and Tempero, E.: 2002, A lightweight web-based case tool for sequence diagrams, *Proceedings of SIGCHI-NZ Symposium On Computer-Human Interaction (CHINZ 2002)*.

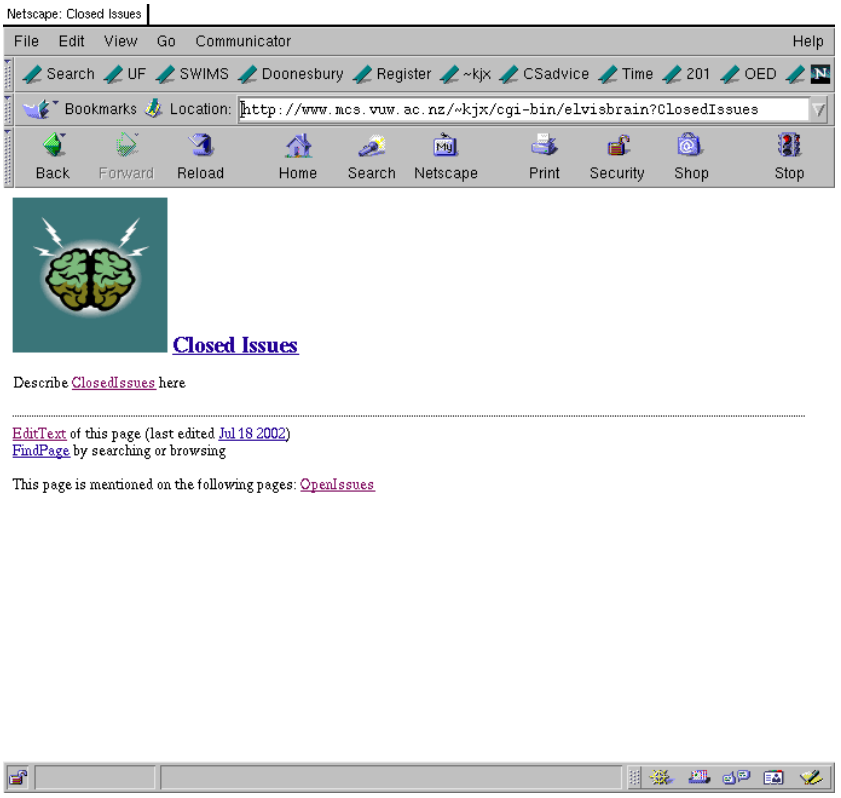


Figure 15: A blank, newly-created WikiWikiWeb page, ready for editing.

Leuf, B. and Cunningham, W.: 2001, *The Wiki Way*, Addison-Wesley.

Mys: n.d., Mysql, http://www.mysql.com.

Noble, J. and Biddle, R.: 2002, Notes on postmodern programming, *in* R. P. Gabriel (ed.), *Proceedings of OOPSLA 2002 Onward! Session*.

Sun Microsystems Inc.: 2002, Javaserver Pages White Paper, *java.sun.com* .

Wall, L., Christiansen, T. and Orwant, J.: 2000, *Programming Perl*, 3rd edn, O'Reilly.