Design Patterns with Use Case Maps:

A Case Study in Reengineering an Object-Oriented Framework

SCE 95-17

R.J.A. Buhr, R.S. Casselman, T.W. Pearce

18 June 1996

Department of Systems & Computer Engineering
Carleton University
Ottawa Canada K1S 5B6
email: buhr@sce.carleton.ca
tel: (613) 520-5718
fax: (613) 520-5727

Design Patterns with Use Case Maps:
A Case Study in Reeingineering an Object-Oriented Framework

R.J.A. Buhr, R.S. Casselman, T.W. Pearce

Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive
Ottawa, Ontario, Canada, K1S 5B6
email: buhr@sce.carleton.ca
phone: (613) 520 5718
fax: (613) 520 5727

Abstract

We show how a new technique called *use case maps* helps humans to understand, capture, analyze, reuse, and change high-level behaviour patterns in complex software. We do so through the example of reengineering an object-oriented framework, a type of software that is well known to be difficult to understand. In the framework we studied (HotDraw), we trace scenarios through the software and express the behaviour patterns we see with use case maps. These maps express high-level behaviour patterns in terms of cause-effect paths, above the detailed level of messages. We find some unexpected, irregular patterns in Hotdraw's use case maps, redraw the maps, and make resulting changes to the software. The contribution of this paper is intended to be, not changes to Hotdraw, but the techniques we used to aid human understanding and communication of high-level behaviour patterns.

Key words: design patterns, reverse engineering, reengineering, use cases, use case maps, frameworks.

1.0 Introduction

This paper applies a new technique called *use case maps* [2] to pattern-guided reengineering, using the example of a public-domain, object-oriented framework called HotDraw [13]. We chose frameworks as the application because frameworks are well known to be difficult to understand [18]. We chose HotDraw because it is publicly available and widely known, not because we thought it needed reengineering; our original purpose was only to show how use case maps could help to understand and document an existing framework, but changed to reengineering when we discovered unexpected, irregular patterns in our use case maps. Although we describe the version of HotDraw we studied (v4.0 dated June 1992) and propose some changes to it, the contribution of this paper is not reengineering of HotDraw v4.0 (which is now outdated), but the techniques we use to aid human understanding and communication of high-level behaviour patterns.

Use case maps are a new, high-level technique for describing behaviour patterns in systems at the level of whole-system architecture (henceforth referred to more briefly as "behaviour patterns"). This use of the term *pattern* may be controversial in the object-oriented community. We use the term in its ordinary English sense to mean a recognizable visual form. However, there are also a number of ways in which the visual forms we shall present are design patterns in the object-oriented sense of the term: for example, they present design solutions in a general and reusable form. However, they are different in level of abstraction, scale, and application from design patterns like those in [7].

Use case maps focus on high-level behaviour, meaning above the level of messages between objects, which is the level of description of behaviour in current practice, as exemplified by [7], [10], and [12]. Use case maps stand well back from messaging details to get an overall view of behaviour patterns in cause-effect terms. Use case maps are related to use cases [12] but use cases are textual and emphasize black-box behaviour of systems, whereas use case maps are visual and trace use case paths from external stimuli through internal components of systems.

Use case maps have four main elements: *paths* that trace the progression of causes and effects from points where stimuli occur (usually, but not necessarily, in the environment), through the components of a system, to points where responses are felt; *responsibilities* that link paths to components; *couplings* between paths that connect them together into larger patterns; and *components* that perform responsibilities. Components include familiar ones like objects and teams of objects, but also unfamiliar ones called *slots* that may be occupied in a dynamic fashion by objects as scenarios unfold (these are *not* the same as the slots of prototype-based languages). Slots enable us to express, with fixed use case maps, the essence of structural dynamics (the creation, changing visibility, and destruction of objects as execution proceeds) at a level of abstraction above the intricate details that are used to achieve it in software.

The body of the paper focuses on the application of use case maps to the problem we studied, in a way that should be readable without referring to separate descriptions of notation. Appendix A summarizes the elements of the notation used in the body (see [2] for the complete notation).

2.0 What We Did

Reengineering has some well defined steps [4] which we followed as outlined below (refer to Figure 1).

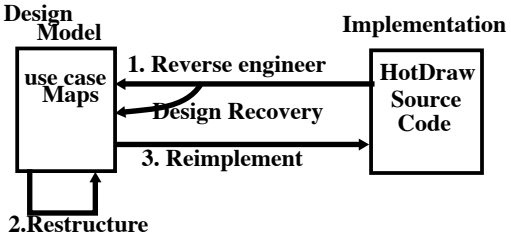


Figure 1: The Reengineering Process

Reverse Engineering: HotDraw (described in Section 3.0) was reverse engineered (Section 4.0) from its source code into a use case map representation. The only tools we required for this study were the Smalltalk execution environment and freehand drawing tools. The major reverse engineering activities were reading source code and source code comments, and stepping through the Smalltalk debugger to follow execution traces. Reverse engineering involved an element of *design recovery*, i.e., the use of experience and intuition in addition to existing implementation artifacts, like source code comments, to produce the design abstractions. Programmers may have high level patterns in mind but sometimes take shortcuts, so sometimes we had to guess at patterns the programmer had in mind. This kind of thing makes automating the process of reverse engineering difficult. During design recovery, we observed unexpected, irregular patterns in the use case maps.

Restructuring: The recovered design was then restructured (Section 5.0) at the use case map level into more regular and understandable patterns than we found in the code. The abstract view provided by use case maps helped us to rise above intricate detail at the level of interobject

messaging, visibility, and dependency, to spot places where restructuring would make the patterns more uniform and understandable.

Reimplementation: The HotDraw source was then reimplemented (Section 6.0) using the maps as a guide. Here, we employed use case maps as a reference, to guide writing code and to make sure paths in the maps were being followed while stepping through code with a debugger. The maps helped to resolve issues affecting information hiding and separation of concerns [15], including: the need for new classes, the allocation of responsibilities between objects, and the visibility relationships between objects.

3.0 An Overview of Hotdraw and Use Case Maps

This section introduces both the application we studied (HotDraw) and the main technique we employed for the purpose (use case maps). Figure 2 shows the user interface of HotDraw. The tools in the tool palette include a selection tool, a scroll tool, an eraser tool, and various figure creation tools. When selected, a tool becomes the current tool and is highlighted in the tool palette area. The current tool controls the interpretation of user actions (e.g., mouse clicks) in the drawing area. A figure or figures may be selected with the mouse using the selection tool. A selected figure has handles displayed on it that may be manipulated to change the figure's attributes, e.g., size and colour. The current group of selected figures are the focus of operations such as cut, copy, delete, group and ungroup. These operations are selected from a popup menu in the drawing area. A set of figures is called a drawing; drawings may be saved to and loaded from disk using the popup menu in the drawing area. The name of the current drawing is displayed in the name stripe.

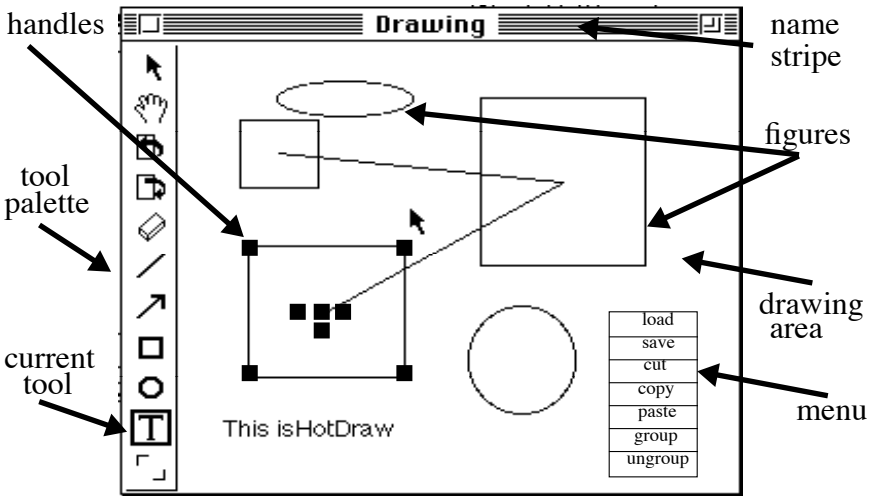
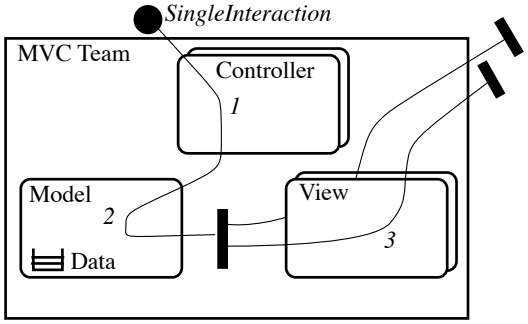


Figure 2: HotDraw User Interface

Figure 3 employs a use case map to give a high-level view of the MVC pattern ([9], [14]) around which HotDraw is organized. In HotDraw, one MVC team is used to manage each of the three different areas of the screen of Figure 2 and the three teams are coupled together to manage the system as a whole. It is at the level of trying to understand high-level behaviour patterns in the system of coupled MVC teams that use case maps are particularly helpful; however, Figure 3 shows only one generic MVC team, to introduce the ideas. Figure 3 shows three out of of four elements of use case maps, namely paths, responsibilities and components; couplings come later when we put the pieces together (Figure 5 and Figure 7). The component labelled `MVCTeam` is a team, meaning it indicates an operational grouping. The components labelled `Model`, `View` and

Controller are objects, meaning they provide data or procedural abstractions. The Model object maintains the semantic data of an application, e.g., the figures in a drawing editor, the View objects display the data of the Model objects on output devices, and the Controller objects interact with users through input devices. The component labelled Data is a pool, meaning a place for keeping any kind of data (including, later on, objects to go into slots elsewhere, e.g., see Figure 5 and Figure 7) The positioning of the Data pool inside the Model object means that Model manages it. In general, the data of Model objects may be displayed in different ways, therefore Model may be associated with several Views. This is shown in the figure by an overlapped set of replicated View objects called a stack. There is a corresponding stack of Controllers because Controllers are paired with Views (this pairing is not a formal property of the map, but a property of the pattern that must be identified separately). The *SingleInteraction* path traces a scenario in which a user interacts with a direct-interaction device; the Controller handles the user request responsibility *(1)*, the Model has the responsibility of changing the data *(2)*, and the Views have the responsibility of updating their displays *(3)*. The fork pattern (called an AND fork) indicates that changes to the model must be propagated to all views, without indicating how this is to be done with messages. The postconditions are that the Model's data has changed and new data has been displayed on the screen in each of the Views. Even if an MVC team has only a single View, instead of a stack of them as here, the AND fork pattern may be needed to propagate changes to other MVC teams.



Responsibilities
1. Handle user request
2. Change data
3. Update display

Figure 3: A Model-View-Controller Use Case Map

Other MVC behaviour patterns than the one in Figure 3 are possible in practice, but for purposes of this paper we shall refer to this one as the standard pattern. There may be detailed variations to the standard pattern. For example, a user may have to interact several times with the Controller during a single scenario, say twice, once to request a menu and a second time to choose an item from the menu, or three times, if a menu selection pops up a dialogue box for the user to fill in. Because responsibilities may be as large-grained as desired, these variations may be covered by a single map of the form of Figure 3 by making a suitably large-grained interpretion of responsibility *(1)*. These variations may also be expressed more explicitly by showing the path going back and forth between the user and the Controller (e.g., some paths in Figure 5 and Figure 7). There is no implied preference of one of these ways over the other, it depends on how much detail you want to make visible in a map. We only caution the reader against using back-and-forth traversals in use case maps to represent details at the level of interobject messaging, which are better left to message sequence charts.

A very important point of this paper is that use case maps may be used to pinpoint places where other kinds of design patterns may be used, without having to specify details associated with these patterns in the maps. For example, in Figure 3 the model-to-view AND fork pinpoints a place where Smalltalk's dependency mechanism could be advantageously used to propagate model

changes to views. This dependency mechanism is a design pattern at the level of classes and messages that enables dependent objects to register themselves with source objects and source objects to send notification messages to all dependent objects (the Observer pattern in [7] is a generalization of it). In Smalltalk, classes inherit methods *addDependent*, *removeDependent*, *changed*, and *update* from the root class. The *changed* method sends the message *update* to all dependents; objects send themselves *changed* messages to trigger this whenever needed (the actual protocols are richer than this, but this is the essence). In an MVC team, `Models` would act as sources and `Views` act as dependents in a dependency relationship.

The separation of high-level patterns in use case maps like Figure 3 from the details of the dependency mechanism (the observer pattern) is advantageous because it enables us stand back from intricate details that can easily obscure the big picture. Some reasons why the details may obscure the big picture are as follows: dependencies may change as the execution unfolds, notifications may be propagated through chains of dependencies, and the places in the code where all of this is made to happen are scattered among many classes and methods. Here is an example of the way use case maps stand back from the details: If, in Figure 3, the views are to be registered as dependents of the model, then messages must flow from the views to the model to do so. There are no view-to-model paths in the figure corresponding to such messages. Such paths are not necessary because the need for the messages can be inferred from the fact that the dependency mechanism is prescribed. Figure 3 is a simple case in which all dependencies may be registered before the paths start (because there are no slots in the figure, all dependencies may be assumed to be fixed for the duration of the paths), but the rule that extra paths are not necessary applies even for cases in which dependencies may be registered during the course of a path (e.g., because objects are moved into slots along the path, as at *A8* in Figure 7).

We hasten to add that not all design patterns at the level of abstraction of the dependency mechanism (observer pattern) can be related to use case maps in this way, only ones that have a behavioural aspect.

4.0 Reverse Engineering and Design Recovery

This section presents a use case map of HotDraw that resulted from our reverse engineering and design recovery exercise. The source code comments of HotDraw suggest a three-team MVC organization (Figure 4), as follows):

- *In class DrawingController*: “A DrawingController is the Controller of a pair of MVC triads. A Drawing and a DrawingView is one pair and a DrawingEditor and the same DrawingView is the other pair. A DrawingController’s primary task is to delegate mouse and keyboard activity to the current Tool of the Drawing.”
- *In class ToolPaletteController*: “ToolPaletteController is the Controller responsible for detecting the change of Tools when a new Tool is selected in the ToolPalette.”
- *In class ToolPaletteView*: “ToolPaletteView is the View responsible for the displaying and highlighting of Tools in the Palette ToolBar.”

The first comment above suggests the presence of what we call the `EditorMVC` and `DrawingMVC` teams. The last two comments suggest the presence of what we call the `ToolMVC` team. To step back from the code details to get a high-level view, we adopted the convention in Figure 4 of showing all members of all MVC teams as distinct components. HotDraw actually shares some objects between MVC teams, meaning that a single object may play the same role (controller, model, or view) in more than one MVC team. We adopt the convention of not

showing this sharing in component terms, because the resulting patterns in use case maps give useful insights into implementation problems arising from the sharing (see Section 4.2). Here are the highlights of the teams in Figure 4:

- *EditorMVC*. The `EditorMVC` team manages the sets of figures, called drawings, created with the tool. This includes loading a drawing for editing and storing a drawing away. The team is responsible for the load and save options of the menu in the drawing area of Figure 2.
- *ToolMVC*. The `ToolMVC` team manages interactions in the tool palette area. Users select a new tool with the mouse. The selected tool becomes the new current tool (in the figure, the existence of a current tool is indicated by a `CurrentTool` slot internal to the `DrawingEditor`). The `DrawingEditor` is shown in abstract form as a team, although it is implemented as a Smalltalk object, to conform to the rule of no decompositon of objects in use case maps (see Appendix A). The pool is not inside `CurrentTool` (in contrast to the pools inside `DrawingEditor` and `CurrentDrawing`) because `CurrentTool` is the destination of objects from the pool, not the manager of the pool.
- *DrawingMVC*. The `DrawingMVC` team manages user operations on the figures that are currently selected (i.e., have handles displayed on them—note the `Handles` pool in the view). The team operates in the drawing area of the tool and is responsible for the menu operations cut, copy, paste, group and ungroup. The `CurrentDrawing` part of the `DrawingMVC` is a slot because different drawings may be edited at different times. This slot is filled by a drawing object from the pool in the `DrawingEditor` of the `EditorMVC` team; each drawing object manages its own `Figures` pool.

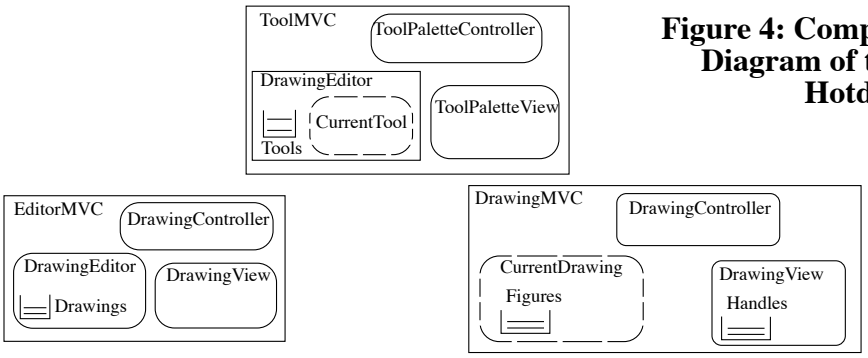


Figure 4: Component Context Diagram of the Original Hotdraw

4.1 Use Case Map of the Original HotDraw

Figure 5 shows our use case map of the original HotDraw. The paths follow a set of use cases in which a user selects a drawing to edit, selects the selection tool from the tool palette, and selects a figure with the selection tool. There are many other use cases we could describe, e.g., cut a figure, save a drawing, resize a figure, etc. However, the ones we have chosen cover the most important paths. Specifying the complete behaviour of a system is not the intent of use case maps, only covering the paths that will most influence major design decisions. This map is concerned only with the behaviour of the MVC teams, not with that of the Figures and Handles objects that appear on the screen; the latter appear in the map only indirectly, as managed data in the `Figures` and `Handles` pools. Patterns at the level of these objects are regarded as detail in relation to this map (although they could be included—see [2]).

Before examining this map as a whole, let us trace the *LoadDrawing* path in it to give a sense of the nature of the map (see Appendix A for notational details). The other paths can be understood in similar terms. The *LoadDrawing* path begins with a mouse press in the drawing area. At *A1*, the

`DrawingController` displays a menu. At *A2*, the user chooses the load option from the menu. At *A3*, the `DrawingController` decodes the menu selection. At *A4*, the `DrawingEditor` displays a dialogue box to the user. At *A5*, the user enters the name of a drawing to edit in the dialogue box. At *A6*, a drawing object is moved from the pool to the path (a new drawing object is created and moved into the pool first, if necessary). The drawing object is imagined to flow along the path from this point on (because this type of move is aliased, it also stays in the pool as data). At *A7*, the `DrawingView` object of the `DrawingMVC` team is informed of the change to the current drawing and the display is updated. At *A8*, the new current drawing is moved into the `CurrentDrawing` slot.

Standing back from the details of Figure 5, observe that there are two interpath couplings in this map, one in the `ToolMVC` between the *SelectTool* and *SelectFigure* paths and the other in the `DrawingMVC` between the *LoadDrawing* and *SelectFigure* paths. In both cases, the paths are coupled through slots, the coupling is between one path moving an object into the slot and the other path using the object to perform responsibilities, and the coupling occurs through a postcondition of one path (the slot is occupied) becoming a precondition of the other.

There are some surprises in Figure 5 relative to the expectations we have formed from Figure 3 (the surprises are spotlighted in the skeleton map shown in Figure 8(a)): (1) Why is the view part of the `EditorMVC` not touched by any path? (2) Why does the *LoadDrawing* path take an excursion from the model part of the `EditorMVC` into the environment to interact with the user, instead of from the controller part? (3) Why does the *LoadDrawing* path go from the view part to the model part of the `DrawingMVC`, instead of the other way round? (4) Why does the *SelectFigure* path take a detour to `ToolMVC` along the otherwise-standard controller-to-model route in `DrawingMVC`? (5) Why does the *SelectTool* path go back to the controller in the `ToolMVC` to highlight the selected tool, instead of doing it in the view part? (6) Why does the view part of the `DrawingMVC` manage what appears to be model data, namely the `Handles` pool? These surprises turn out to be the results of detailed decisions taken at the code level. One issue that led to them is elaborated in Section 4.2 and others are discussed later in relation to the properties of a restructured map (Figure 7, Figure 8(b)).

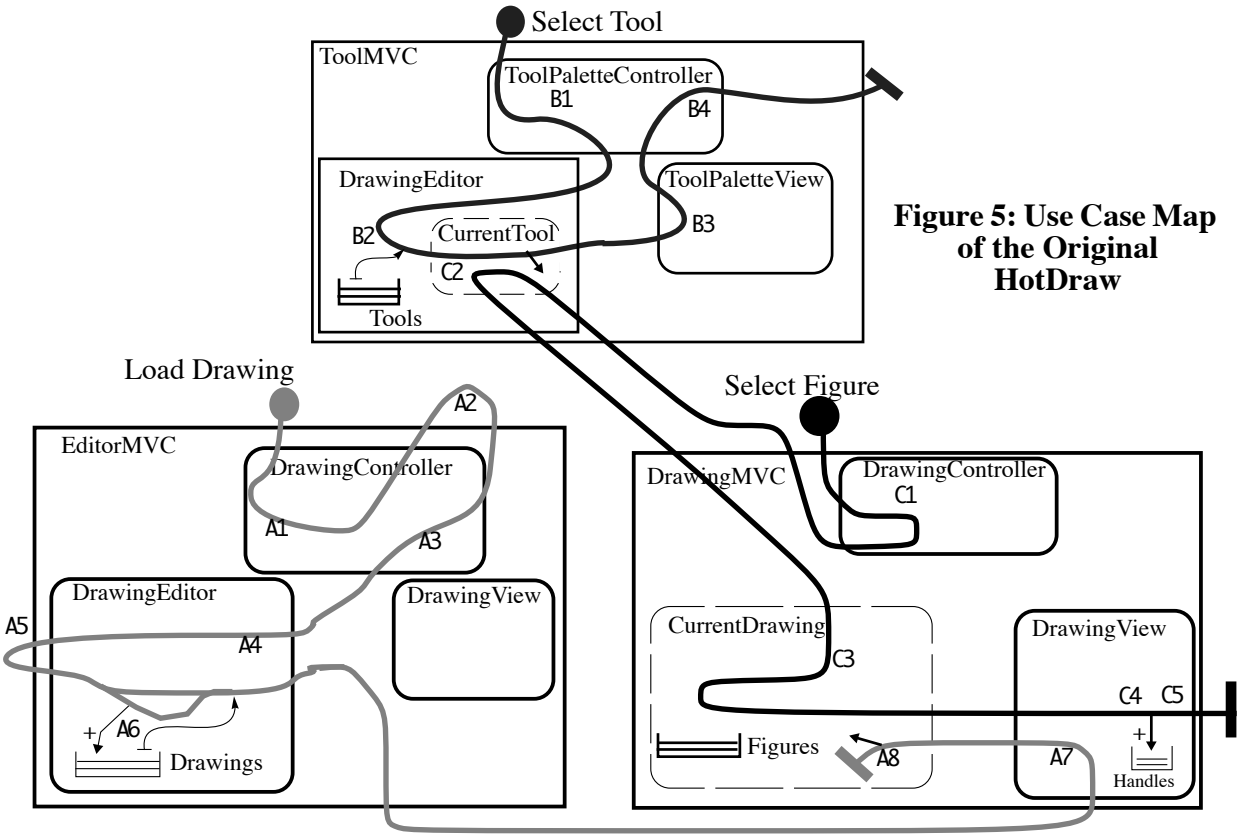


Figure 5: Use Case Map of the Original HotDraw

Path *Load Drawing* Responsibilities. *C1*. inform current tool of mouse press; *C2*. react to mouse press. *C3*. find figure at cursor position; *C4*. create handles for the selected figure; *C5*. display handles.**.** *A1*. display menu; *A2*. choose menu item (load); *A3*. decode menu item; *A4*. display load drawing dialogue box; *A5*. choose drawing to edit; *A6*. create new drawing or select existing drawing and add to the use case path; *A7* update the drawing view; *A8*. install new current drawing.

Path *Select Tool* Responsibilities. *B1*. handle mouse press in tool palette; *B2*. find tool at cursor and put in CurrentTool slot; *B3*. receive update; *B4* highlight the currently selected tool.

Path *Select Figure* Responsibilities. *C1*. inform current tool of mouse press; *C2*. react to mouse press. *C3*. find figure at cursor position; *C4*. create handles for the selected figure; *C5*. display handles.

4.2 Effects of Object Sharing

The detour taken in Figure 5 by the *SelectFigure* path to `ToolMVC` along the otherwise-standard controller-model route in `DrawingMVC` is a side effect of object sharing in the implementation, as will now be explained. Figure 6 uses filled shapes superimposed on pairs of objects in the different HotDraw MVC teams to identify object sharing as it appears in the implementation. Each shape indicates that a single concrete object is shared between two different teams. For example, the controller parts of the `EditorMVC` and `DrawingMVC` teams are implemented by one concrete object (Drawing Controller).

Object sharing can easily lead to unwanted dependencies in a system due to a lack of information hiding between teams. *Dependencies are introduced when a shared object uses information to which it is privy in one role to fulfil its responsibilities in another.* The anomalous detour in the *Select Figure* path arises because of a chain of such dependencies. The first link in the chain is the

single `DrawingController` object using knowledge privy to its role in the `EditorMVC` team to fulfil its role in the `DrawingMVC` team (by asking the `DrawingEditor` for the object in `CurrentTool`). The second link in the chain is the single `DrawingEditor` object satisfying this request by using knowledge of `CurrentTool` that is privy to its role in the `ToolMVC` team. How the source code does this is so simple and innocent-looking that it could easily go unnoticed in a code review, but its ramifications are not so simple. Separation of concerns has been violated because the `DrawingEditor` is expected to know the current tool, when `the Model` of the `EditorMVC` team should know nothing about tools. Information hiding has been compromised because the `DrawingController` uses information internal to the `EditorMVC` and `ToolMVC` teams to gain access to the current tool.

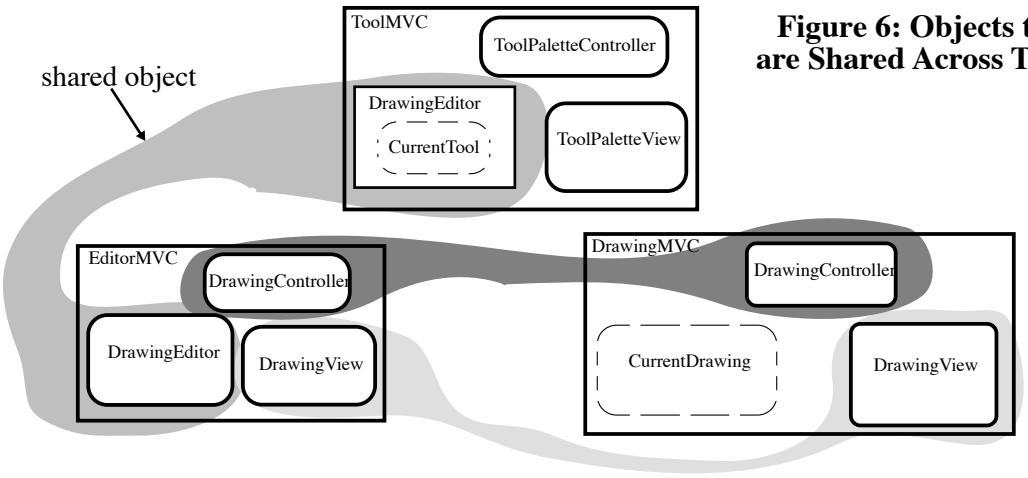


Figure 6: Objects that are Shared Across Teams

5.0 Restructuring

We restructured the map to give a more regular high-level picture (Figure 7) and then used this map to guide the design and implementation of details. All paths now follow the standard pattern of Figure 3 with variations of detail but not strategy; in other words, all follow the control-model-view path sequence. The motivation for this restructuring is not mindless obedience to an arbitrary standard (Figure 3), but understandability of the code, which affects other “ilities” such as maintainability and evolvability. Standard patterns enable us to have confidence that we understand “how it works” as a whole, even when examining and modifying small parts of “it” (the code).

Before examining the changes in detail, let us trace the *LoadDrawing* path of Figure 7 to give a sense of the nature of the map in comparison to the original map (compare this with the earlier description of the same path in relation to Figure 5). The path begins with a mouse press. At *A1*, the `MenuBarController` displays a menu. At *A2*, the user chooses the load option from the menu. At *A3*, the `MenuController` decodes the menu selection. At *A4*, the `DrawingModel` displays a dialogue box to the user. At *A5*, the user enters the name of a drawing to edit in a dialogue box. At *A6*, a drawing object is moved into the path (as well as staying in the pool as data). At *A7*, the name of the new current drawing is displayed in the name stripe of the tool. At *A8*, the new current drawing is moved into the `CurrentDrawing` slot (implying all visibilities and dependencies are established to enable the object to perform its role in the `DrawingMVC`). At *A9*, the figures of the new current drawing are displayed in the drawing area.

To understand the changes, look at Figure 8, which shows the maps of Figure 5 and Figure 7 side by side, with all the textual cues removed and the changed areas spotlighted and numbered for reference. The numbers correspond to the numbered questions we asked earlier concerning Figure 5, which will now be answered; the spotlighted areas in (a) indicate the problems and the correspondingly numbered ones in (b) the solutions. Here is a summary of the spotlighted problems and solutions:

1. *Original Problem* . Why is the view part of the `EditorMVC` not touched by any path? The answer is that, operationally, there is no view object in `EditorMVC` (we showed one only because the code comments said it was an MVC team). The code does not propagate model-to-view changes along the *LoadDrawing* path in a way that can be identified as belonging within `EditorMVC`; all screen changes associated with the *LoadDrawing* path occur through the view object in DrawingMVC. This makes it difficult to justify the characterization of `EditorMVC` as an MVC team.

   Restructured Solution. There is a new class and object (`MenuBarView`) that are responsible for displaying the name of the current drawing in the name stripe of the tool on the screen.

2. *Original Problem*. Why does the *LoadDrawing* path take an excursion from the model part of the `EditorMVC` into the environment to interact with the user, instead of from the controller part? The answer is that the code apparently takes a shortcut here, compared to the standard MVC pattern.

   Restructured Solution. We introduced a slot called `DialogueBox` internal to the `MenuBarController`. The `MenuBarController` now displays the dialogue box to the user. The slot is filled dynamically during an initialization phase (not shown in the map).

3. *Original Problem*. Why does the *LoadDrawing* path go from the view part to the model part of the `DrawingMVC`, instead of the other way round? The answer is that the code does not use the dependency mechansim to propagate model changes to views in this MVC team, but takes a shortcut to update the view directly. This is not consistent throughout the program; sometimes the dependency mechanism is used, sometimes not (as here).

   Restructured Solution. The new map follows the standard MVC pattern of Figure 3 everywhere, with the dependency mechanism prescribed for all notifications. For example, in the implementation resulting from this map, the `DrawingController` object is registered as a dependent of the `ToolModel` object, the `DrawingView` and `MenuBarView` objects are registered as dependents of the `DrawingModel` object, and the `DrawingView` object is registered as a dependent of the object currently occupying the `CurrentDrawing` slot (the latter done dynamically).

4. *Original Problem*. Why does the *SelectFigure* path take a detour to `ToolMVC` along the otherwise-standard controller-to-model route in `DrawingMVC`? The answer is, as was explained earlier, that object sharing was used to take shortcuts. According to a comment in the source code, “A DrawingController’s primary task is to delegate mouse and keyboard activity to the current Tool...”; however, this path is not the only way of doing that.

   Restructured Solution. The slot `CurrentTool` was moved from the `ToolMVC` team to the `DrawingController` of the `DrawingMVC` team.

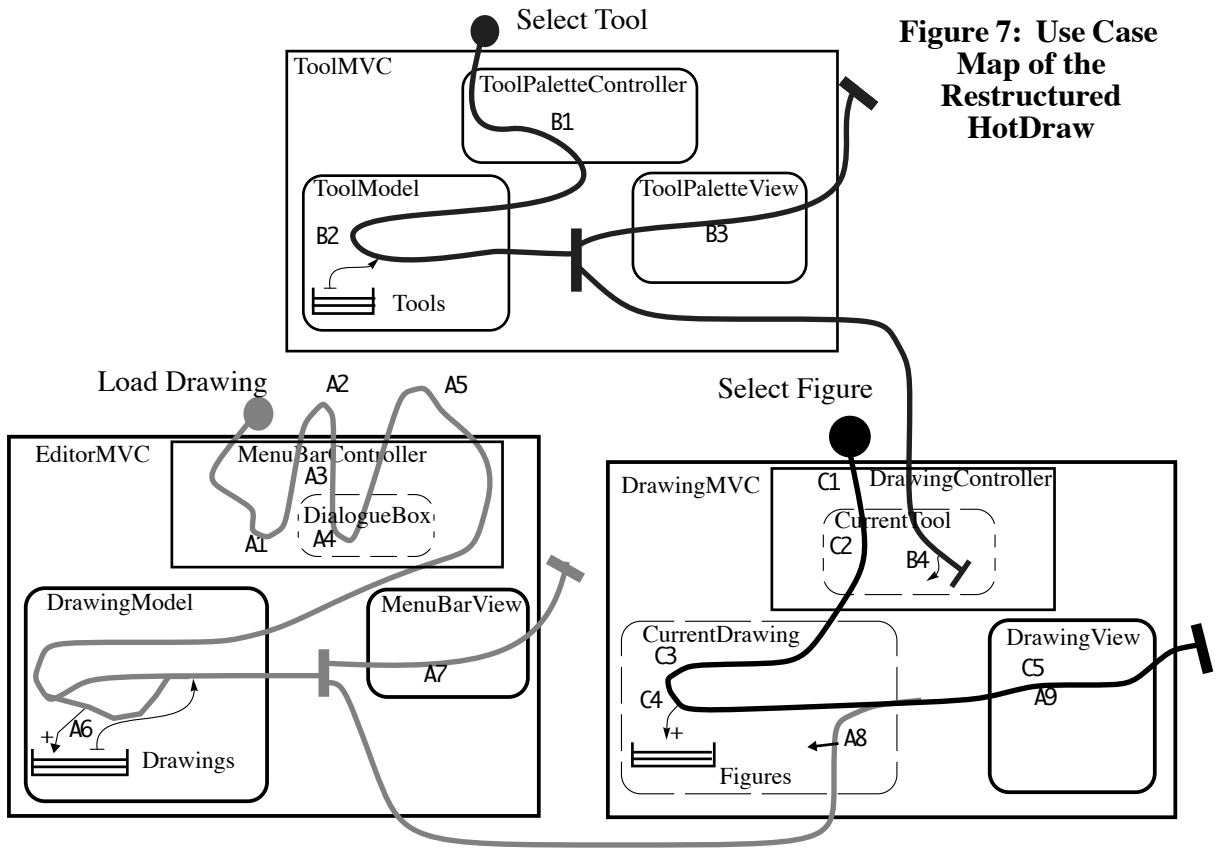


Figure 7: Use Case Map of the Restructured HotDraw

Path *Load Drawing* Responsibilities. *A1*. display menu; *A2*. choose menu item (load); *A3*. decode menu selection; *A4*. display dialogue box; *A5*. choose drawing to edit; *A6*. create new drawing or select existing drawing and add to the use case path; *A7*. update name stripe; *A8*. install new current drawing; *A9*. display figures.

Path *Select Tool* Responsibilities. *B1*. handle mouse press in tool palette; *B2*. find tool at cursor; *B3*. highlight selected tool; *B4* install new current tool (figure selection tool in this example).

Path *Select Figure* Responsibilities. *C1*. inform current tool of mouse press; *C2*. react to mouse press. *C3*. find figure at cursor position; *C4*. create handles for the selected figure and add to the figures pool; *C5*. display figures.

5. *Original Problem*. Why does the *SelectTool* path go back to the controller in the `ToolMVC` to highlight the selected tool, instead of doing it in the view part? This violates the assignment of responsibilities to objects in the standard MVC pattern.

   Restructured Solution. The responsibility was moved to the expected place, i.e., `ToolPaletteView`.

6. *Original Problem*. Why does the view part of the `DrawingMVC` manage what appears to be model data, namely the `Handles` pool? There can be arguments pro and con on the answer to this question. *Pro:* Handles are not permanent properties of figure objects so they are view data, not model data. *Con:* Handles need to be saved with deleted objects to support undo, so they are model data.

   Restructured Solution. Handles are now managed in `CurrentDrawing` (the model). This makes paths for operations not shown, like cut, copy, and undo, more uniform.

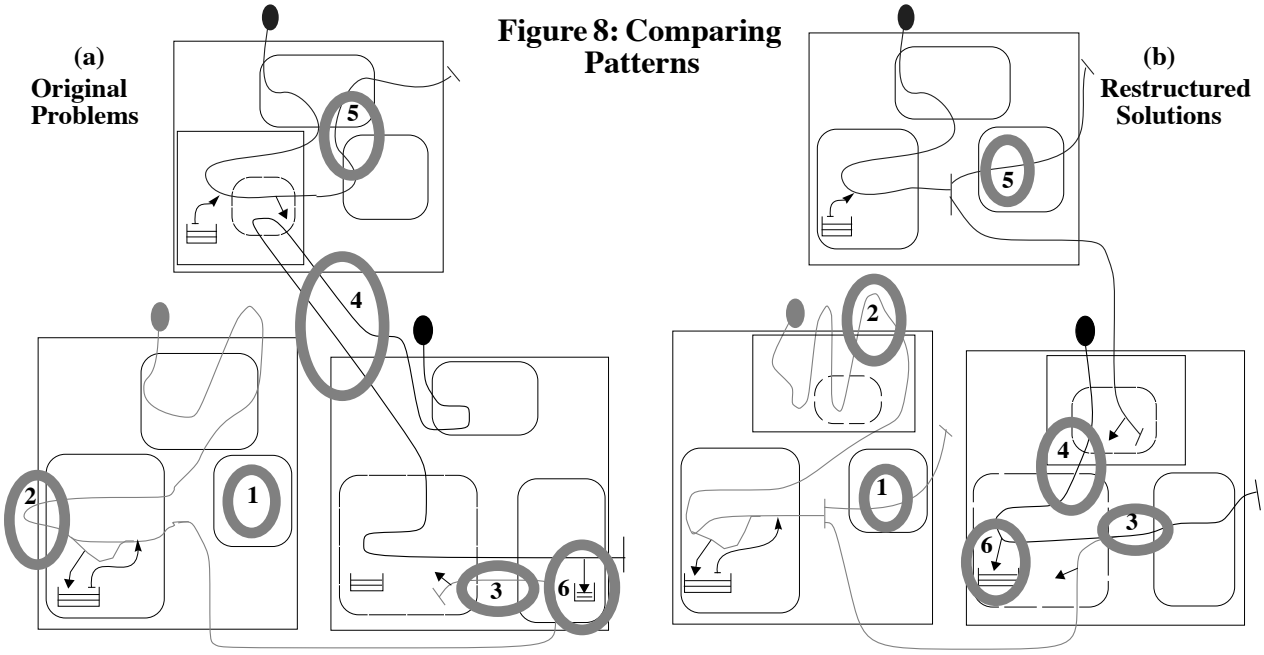
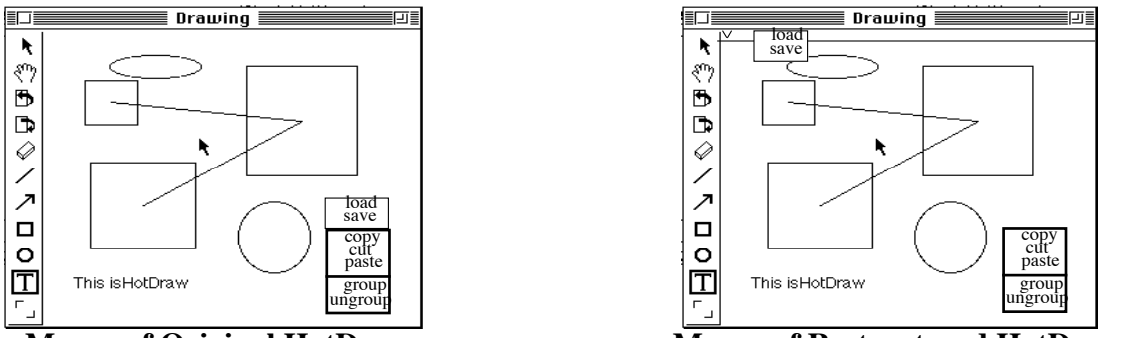


Figure 8: Comparing Patterns

5.1 Effect of the Changes on the User Interface

Some of our proposed restructuring required us to redesign the user interface of HotDraw. The tool has the same functionality as the original, but requires a new screen area with its own menu. This has a number of beneficial effects, as will now be described (Figure 9).



Menus of Original HotDraw **Menus of Restructured HotDraw**

Figure 9: Menus of Original and Restructured HotDraw

Figure 9 illustrates the difference between the two interfaces. In the original version of HotDraw, the save and load operations are part of the pop menu in the drawing area. In the restructured version of HotDraw, we have added a menu bar above the drawing area that has a single menu with the save and load operations in it.

Why the difference? From an internal perspective, the two menus in the restructured version of HotDraw clearly separate the operation of the `EditorMVC` team (responsible for the save and load operations) from the operation of the `DrawingMVC` team (responsible for the copy, cut,

paste, etc. operations). Each team operates in its own rectangular screen area with its own controller. From a user interface perspective, we justify the separation on the grounds that save and load operate on entire drawings, while cut, copy, paste, etc. operate on the currently selected figures. In the original version of HotDraw, a controller object is shared across the `EditorMVC` and `DrawingMVC` teams. In the new version it is easier to have the operations for both teams in the same menu.

From a user perspective, the modified interface is in line with the common convention that file operations are separated from screen operations in the human interface. From the perspective of human understanding of the code, we think the restructured version is easier to understand on all counts.

6.0 Reimplementation

We reimplemented HotDraw by modifying the existing source code while using the restructured use case map as a guide. The reimplementation includes all elements in our use case maps, but is not complete to the extent that we could ourselves release a new version of HotDraw (that was not the objective of the exercise); however, we have done enough of it to satisfy ourselves that the changes we propose are practical. We began by introducing the new classes that would allow us to break the sharing of objects across teams. This required three new classes: `ToolModel`, `MenuBarView`, and `MenuBarController`. We then introduced the necessary dependency mechanisms between objects to support the paths of the maps. Other more detailed modifications included moving blocks of code between classes to support the reallocation of responsibilities (e.g., the tool highlighting behaviour was moved from the `ToolPaletteController` class to the `ToolPaletteView` class) and introducing or removing instance variables (e.g., a currentTool instance variable is needed in the `DrawingController` because it is now the container of the current tool). The final step was to introduce MVC classes for the MVC teams of the use case maps. We found this made the initialization code of the tool more readable.

7.0 Discussion

We employed the following two interesting ways of working with use case maps in this case study. *Discovering irregular patterns:* Because visual patterns in use case maps are easily recognizable by humans, to an extent that is not possible with more linear and detailed message sequence charts, we found that irregular patterns stood out. We then interpreted these as due either to faulty understanding of the code or possible flaws in its design or implementation. When we were unsure, we found it useful to have one team member explain the map to someone else who was following the code. When the map remained irregular, no matter how hard we tried, we concluded that some restructuring might be beneficial. We then used the discovered map to guide restructuring of information hiding units (teams), visibility relationships, and the map itself. *Walkthroughs:* We employed use case maps as references during walkthroughs in the following way. After believing we had implemented a path, we would step through the result, colouring the path segments that the messages covered as we went along. In this way, we performed manual coverage analysis.

Tools supporting use case maps would have been useful, e.g., to draw and manage maps, link responsibilities in the maps to methods, and animate the maps so that the designer can verify that the message sequences achieve the paths in the map (we did this manually in our study). Could a tool automatically reverse engineer code into use case maps or translate use case maps into complete implementations? We think not, at least for the forseeable future, because of a large semantic gap between the maps and code that must be bridged by human thinking, and because the responsibilities

along the paths in the maps are (deliberately) not formal.

Do the techniques scale up? HotDraw is a relatively small framework (37 classes and around 3000 lines of commented Smalltalk code) and this exercise certainly does not prove that the techniques will scale up to larger examples. However, it has sufficient complexity to illustrate the ideas, containing as it does a richness of fine-grained detail that obscures the high-level picture. That use case maps themselves have no inherent scale is obvious by inspection. That cause-effect paths will scale up better than message sequences is plausible simply because they are less detailed. Beyond that, we can offer only our own opinion that the techniques do scale up (see [2] for more).

Do the techniques have wider applicability than the type of program we studied? Although not discussed in this paper, we suggest that use case maps are as useful for expressing high-level patterns in real-time and distributed systems as for ones in object-oriented programs, and that they provide conceptual glue to join these areas and to make tradeoffs between the different issues they raise (see [1] and [2] for more).

We have characterized use case maps as “high-level”; other terms are “system-level”, or “architectural”. Whatever term is used, thinking at this level has a place in the design of object-oriented programs and frameworks. We have noticed, as have others [6], that, in object-oriented design, thinking at this level sometimes takes a back seat to issues of reusability and extensibility. In this exercise, we found that a high-level view of behaviour patterns helped us stand back from the intricate details that are sometimes used to achieve reusability and extensibility in actual programs, to bring classical software engineering concepts, such as information hiding, separation of concerns, high cohesion of modules and low coupling of modules, to bear on object-oriented design. We found that modeling structural dynamics in terms of slots, pools, and components moving along paths—an approach unique to use case maps—was helpful for seeing the implications of issues like visibility and code sharing without becoming bogged down in intricate implementation details.

8.0 Conclusions

We have tried to show, by example, how a new technique called use case maps helps humans to understand, capture, analyze, reuse, and change high-level behaviour patterns in complex software. We have done so through the example of reengineering an object-oriented framework, a type of software that is well known to be difficult to understand. In doing so, we think we have demonstrated some useful properties of use case maps, as follows: They provide a high-level view of behaviour patterns that supplements more detailed views like message sequence charts. They make dependencies between different parts of a system explicit at a high level, dependencies that are difficult to find in the source code. They guide the resolution of issues affecting sharing and visibility of components. They are an effective communication medium between team members. They are a useful tool for spotting high-level irregularities that make code difficult to understand, modify and maintain. They provide documentation of high-level patterns to guide a reimplementation. Our examples must be their own evidence for these claims.

Acknowledgements. This work was funded by several sources, principally BNR and NSERC, but also TRIO. Gerald Karam cosupervised a related thesis. Murray Woodside provided helpful comments.

References

[1] R.J.A. Buhr, A. Hubbard,*Understanding the Behaviour of Real-Time and Distributed Ssytems Constructed from Object-Oriented Frameworks*, SCE report—http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss.ps

[2] R.J.A. Buhr, R.S. Casselman, *Use Case Maps for Object-Oriented Systems,* Prentice Hall, 1996.

[3] R.J.A. Buhr, R.S. Casselman, *Timethread-Role maps for Object-Oriented Design of Real-time and Distributed System,* Proceedings of OOPSLA94, Portland, Oregon.

[4] E.J. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", pp. 13-17, *IEEE Software*, January 1990.

[5] D. Coleman et. al. *Object-Oriented Development: The Fusion Method*. Prentice-Hall Object-Oriented Series, 1993.

[6] J. Coplien, In Forward to *Real-time Object-Oriented Modeling*, by B. Selic, G. Gullickson and P.T. Ward, Wiley, 1994.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[8] D. Garlan and M. Shaw, "An Introduction to Software Architecture," in Advances in Software Engineering and Knowledge Engineering, Vol. 1, World Scientific Publishing Company, 1993.

[9] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementatio*n. Reading, MASS:Addison-Wesley, 1985.

[10] O. Haugen, *Object Oriented Message Sequence Charts*, ITU Contribution, Norwegian Computing Center, P.O. Box 114, Blindern, N-0314 Oslo, Norway.

[11] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems," In *Proceedings of OOPSLA/ECOOP'90*, pp. 169–180, ACM/SIGPLAN, Ottawa, Canada, October 1990.

[12] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.

[13] R. E. Johnson, "Documenting Frameworks using Patterns," In *Proceedings of OOPSLA'92*, ACM/SIGPLAN, Vancouver, B.C., Canada, October 1992.

[14] G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming (JOOP)*, Vol. 1, No. 3, pp. 26–49, September 1988.

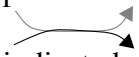
[15] D.L. Parnas, "On the Criteria to be used in Decomposing Systems into Modules", *Communications of the ACM*, Vol. 15, No. 2, pp. 1053-68.

[16] T. Reenskaug, et. al., "OORASS: seamless support for the creation and maintenance of object oriented systems," Journal of Object Oriented Programming (JOOP), October 1992.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ:Prentice Hall, 1991.

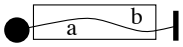
[18] R.J. Wirfs-brock and R.E. Johnson, "Survey of Current Research in Object-oriented Design", *Communications of the ACM*, v. 33, n. 9, pp. 104-124, September 1990.

Appendix A: Use Case Map Notation Employed in This Paper

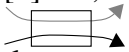
Paths: Paths are identified by a filled circle at the start and a bar at the end. They indicate, in a high-level way, the routes followed by stimuli as they propagate through a system. The paths trace cause-effect sequences (*not* message sequences) from points of stimuli through the components of the system; such sequences are called scenarios. Start points are positioned where stimuli that start scenarios occur (in components or in the environment); end points are positioned where scenarios finish. Different paths may be partially superimposed visually  but their separate end-to-end nature is always assumed to be understood (this may be indicated explicitly in diagrams by different shading or colouring). AND forks indicate that a single path is split into many "concurrent" forks; for sequential programs, the interpretation

is that scenarios take all forks but complete them in some arbitrary order, perhaps by interleaving responsibilities along them. Paths are assumed to have associated preconditions and postconditions for the scenarios that may traverse them.

Responsibilities: Responsibilities

are indicated by labelled points along paths. The position of a responsibility indicates both *where* it is performed (by a component or by users in the environment) and *when* it is performed (in relation to the cause-effect sequence along a path). In general, responsibilities may be coarser-grained quantities than methods or messages. A component may have several responsibilities; we say that the set of its responsibilities identifies the component's *role* in a map. Paths touch components 

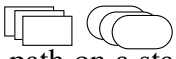
to indicate where responsibilities are performed.

Couplings: In general, paths may be coupled in a variety of ways [2] but, in this paper, the only way is by showing coupled paths touching a shared component  to indicate that the coupling is through responsibilities of the component. For sequential programs, the typical case is that a postcondition of one path in relation to the component becomes a precondition of another path in relation to the same component, e.g., one path moves a component into a slot for another path that requires a component in the slot.

Components: When we use the term "component" in relation to use case maps, we mean an operational entity that can perform responsibilities along paths. Components include teams, objects, stacks of them, slots where they may operate, and pools where they may be held as data.

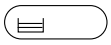
Teams: In use case maps, teams are used to group operationally related components, without committing to whether or not the teams themselves will have explicit existence in the implementation, e.g., as code-level objects with instance variables for team members (the philosophy of use case maps is not to commit to implementation decisions). Teams are also used to indicate "some component", without making a commitment to type.

Objects: In use case maps, objects are primitive components (meaning they have no further structural decomposition into objects at the map level) that are assumed to support procedure or data abstractions through interfaces (although the interfaces themselves, and associated messages, are below the level of the maps). There are no classes in use case maps and no commitment to whether or not map objects are instances of classes in some object-oriented programming language. This separates construction concerns from behaviour concerns. See [2] for ways in which case maps may be used in a coordinated way with class descriptions.

Stacks: A stack  indicates a set of distinct but operationally-identical components. Superimposing a path on a stack implies only one component is touched by the path (which one depends on the preconditions of the scenario). Threading a set of AND forks through a stack indicates that all components are touched, one by each fork.

Slots: Slots are placeholders for dynamic components. They indicate places in maps where dynamic components may become visible and operate for a time in some local context, such as a team. When a slot is occupied by a dynamic component, e.g., an object, the slot is effectively the component. However a slot may contain different components at different times and may sometimes be empty. When we look at use case maps with slots in them, we think of the slots as actual components, while keeping in the backs of our minds that they are only placeholders. The use of slots is a diagramming trick to represent dynamically changing structures with fixed diagrams. Slots are analagous to positions in human organizations that exist independently of their occupancy or occupants. The use of slots may represent quite a step back from the details of code, particulary in a reengineering exercise on code which contains no notion

of slots; this stepping back enables us to get a better high-level view.

Pools: A pool is a place where certain types of components, e.g., objects, are kept as data. Objects from pools go into slots elsewhere in a map to become operational entities there. The positioning of a pool inside a component, such as a team, object, or slot, means that the pool is managed in the context of the containing component (remember, containment in use case maps signifies operational grouping, not necessarily code-level containment). The appearance of a pool inside an object  does not violate the rule of *no structural decomposition* for objects in maps because that rule applies to objects as operational entities, not to objects as data.

Moves: Small arrows touching paths indicate movement of dynamic components into paths from pools or slots and out of paths into pools or slots. This abstract model enables us to step back from the details of the code to see high-level patterns clearly. A "+" beside an arrow indicates *create and move* (such an arrow does not require a source); a "-", not used in this paper, would indicate *move and destroy.* Moves are normally unaliased; aliased moves are indicated by arrows with bars at their tails. Aliasing means the component not only moves to the place at the head of the arrow but also stays in the place at the tail, e.g., moves out of a pool into a path (destined for a slot elsewhere), but also stays in the pool as data (the implication is that any changes occuring while it is operating in a slot will be preserved in the pool).

Visibility and Dependency: Moving a component from a path into a slot implies that all visibility and dependency relationships must be established—by whatever means are required—to enable the component to fulfil its role in the context of the slot. The maps do not show the means. For example, a use case path would *not* go back and forth between the slot and surrounding objects to show that dependency-registration messages must be sent by surrounding objects, because, if they are needed, they are implied by (1) the movement into the slot and (2) the prescription of the dependency mechanism as an implementation pattern.

Terminology: The terminology used here is consistent with [2]. In earlier work [3], we used the terms "timethread maps" or "timethread-role maps" for what we now call use case maps, "timethreads" for what we now call use case paths, and also used other unfamiliar terminology associated with roles ("carrier", "worker") that we no longer use.