Active Programming Strategies in Reuse

Mary Beth Rosson and John M. Carroll

IBM T. J. Watson Research Center
Yorktown Heights, New York 10598, USA

Abstract. In order to capitalize on the potential for software reuse in object-oriented programming, we must better understand the processes involved in software reuse. Our work addresses this need, analyzing four experienced Smalltalk programmers as they enhanced applications by reusing new classes. These were *active* programmers: rather than suspending programming activity to reflect on how to use the new components, they began work immediately, recruiting code from example usage contexts and relying heavily on the system debugger to guide them in applying the borrowed context. We discuss the implications of these findings for reuse documentation, programming instruction and tools to support reuse.

1 Introduction

A key attraction of object-oriented programming languages is the potential they offer for the reuse of software components. A well-designed object class defines a tightly encapsulated bundle of state and behavior that can be "plugged into" a target application to fill some functional need — hence the popular metaphor of a "software IC" [4,5]. And while most of this potential has been asserted rather than demonstrated, empirical evidence documenting the advantages of an object-oriented language for code reuse is beginning to emerge [17]. At this point, however, we know very little about the *process* of component reuse and thus how we might best support reuse activities.

A programmer attempting to recruit existing software components for his or her current project must carry out two basic tasks. First, the candidate component(s) must be identified. This may be trivial in cases where the component was self-generated or is already familiar to the programmer (see, e.g., [6,16]). However, much of the missed potential in software reuse arises in situations where the programmer knows little or nothing about the component in advance. As component libraries increase in size, the difficulty of locating novel functionality increases commensurately. Not surprisingly, researchers have begun to apply a variety of classification and information-retrieval techniques to address the difficult problem of locating unknown functionality within large class libraries [12,21].

Once a candidate component has been identified, the programmer must incorporate the component into the ongoing project. Again, if the component is self-generated or already familiar, this process is simplified: the programmer already knows what it does and how it is used, and merely must apply this knowledge to the new situation. But for unfamiliar components, the programmer must engage in at least some form of analysis, determining what the component does and how it can contribute to current needs, and then designing and implementing the code needed to extract the desired functionality [2, 10]. Researchers are only beginning to explore how one might document code intended for reuse (see, e.g., [14]). But from the perspective of a programmer considering reuse, one requirement is clear: understanding how to use a component must take less time and effort than (re)building the component itself. Indeed, given programmers' general preference for self-generated code, the cost of reusing a component should be considerably less than that of creating it.

This paper seeks to elaborate the requirements for reuse documentation and tool support through analysis of experts carrying out a reuse task. We observed Smalltalk programmers enhancing an application through the reuse of classes we provided. Most generally, our goal was to characterize the strategies and concerns of the programmers as they attempted to reuse the novel classes — by understanding what does and does not work well in the current reuse situation, we can begin to reason about possible modifications or enhancements. More specifically, however, we were interested in the role that examples might play in documenting reusable components. We have been researching example-based programming environments for learning and for reuse [3,13,20,22], and this empirical setting provides an opportunity to examine experts' natural strategies for finding and applying example information.

2 The Reuse Situation

Four experienced Smalltalk programmers participated in the study. All had been programming in Smalltalk/V® PM [8] for over two years, and had over 10 years of general programming experience. All had worked on user interface development in Smalltalk, largely on building components for advanced user interfaces (e.g., multimedia objects, direct manipulation techniques, visual programming).

Each programmer completed two reuse projects, in two separate sessions. The reuse situation approximated the application prototyping activities these programmers carry out in their normal work environment, in that both projects involved an enhancement to the user interface of an already-written interactive application. The applications were simple but non-trivial examples of Smalltalk projects; in debriefing sessions after the experiment, all of the programmers judged that these were representative reuse programming tasks. The order of the projects was counterbalanced — one project served as the first project for two of the programmers, the other as the first for the other two. During their second sessions, programmers were introduced to the Reuse View Matcher [22] and were allowed to use this tool while completing the project. Due to space limitations, this paper will not discuss the second set of sessions involving the Reuse View Matcher.

The programmers were read brief instructions at the beginning of each session, describing the application they were to enhance, and identifying the class they were to reuse in making this enhancement. They were told that they were not expected to spend "more than a couple of hours" on the project and that they should not worry if they did not complete it in this amount of time. Finally, the programmers were asked to "think aloud" while they worked, to vocalize their plans and concerns as they worked as much as possible without interfering with their activities [9].

After hearing these instructions, the programmers were given an extended introduction (approximately 20 minutes) to the application to be enhanced; this involved going over a hierarchical view of the major application classes, a design diagram of application objects and their connections, descriptions of typical interaction scenarios, as well as a comprehensive walk-through of the code. The intent was to familiarize them with the application enough so that their problem-solving efforts would focus on the reuse of the new class rather than on understanding how the existing application worked. No information other than the name was provided about the class to be reused.

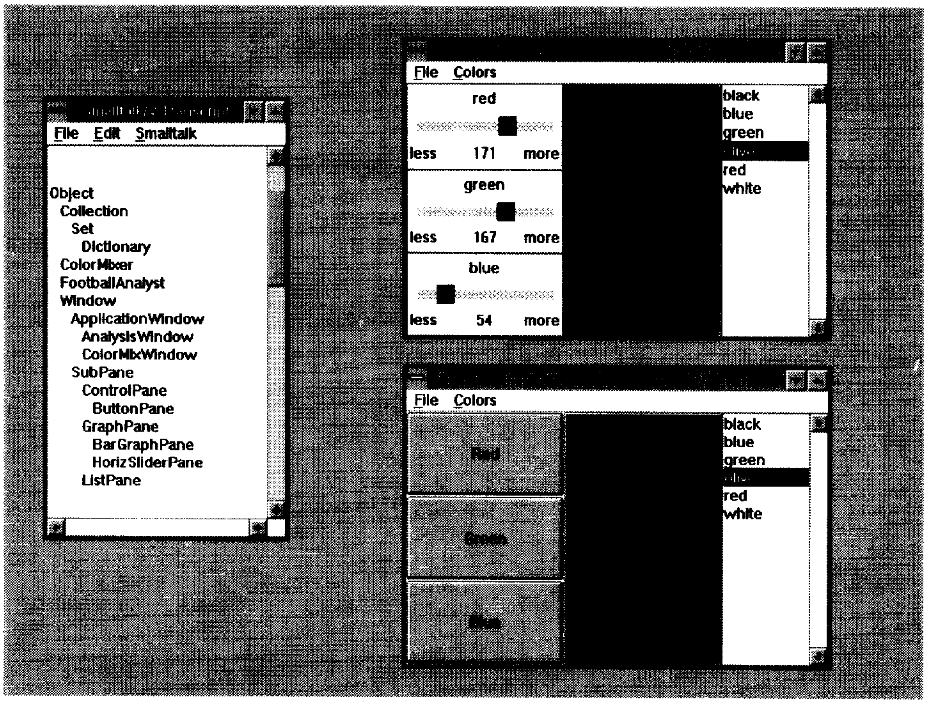
During the reuse task, programmers worked at their own pace in a standard Smalltalk/V PM environment. The experimenters took notes and made videotapes of the programming activity on the display, occasionally prompting the programmer to comment on a plan or concern. All projects were completed within one and a half to two hours.

2.1 The Color-Mixer Project

One of the projects consisted of an enhancement to a color-mixer. The color-mixer converts rgb values input by the user to create custom colors; these colors are stored in and retrieved from a database of named colors. The original application has three buttons for red, green and blue (see Figure 1); clicking one of these buttons brings up a dialog box in which the user types an integer to manipulate a color component. The color being edited is displayed as a "swatch", and is flanked by the list of saved colors. Users can select colors from the list, as well as adding and deleting colors.

Because everything in Smalltalk is an object, and because objects typically inherit a good deal of their functionality, it is difficult to characterize the "size" of applications. However, the most important objects in the color-mixer are instances of six classes (see Figure 1): ColorMixer, ColorMixWindow, ButtonPane, ListPane, GraphPane and Dictionary. The last four classes in the list are components of the standard library. The number of methods in these six classes ranges from six to 54, with an additional 118 to 338 inherited methods.

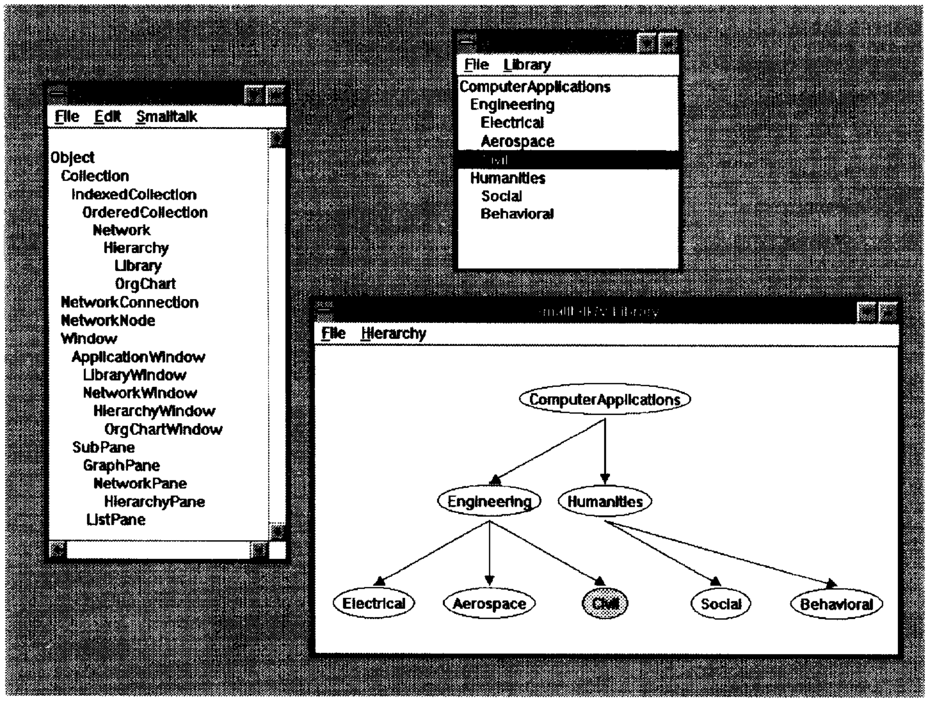
The programmer's task was to replace the button+dialog box input style with horizontal sliders. No information was provided concerning the appearance or functionality of the slider, only that they were to use the new class HorizSliderPane. A typical solution involves the editing of the existing `openOn:` method (this is the method that creates and initializes the windows and subpanes, and the button creation code must be replaced with analogous code for the sliders), and the addition of four new methods (to handle activity in each of the sliders, and to draw any given slider).



```
Object
  Collection
    Set
      Dictionary
  ColorMixer
  FootballAnalyst
  Window
    ApplicationWindow
      AnalysisWindow
      ColorMixWindow
    SubPane
      ControlPane
        ButtonPane
      GraphPane
        BarGraphPane
        HorizSliderPane
      ListPane
```

Figure 1. The Color-mixer Project: On the left is a listing of the major classes involved in the color-mixer and football analyst applications; indentation in the list signifies superclass-subclass relationships. In the upper right is the original color-mixer; beneath it is the application enhanced to use sliders as input devices.

The class library included an example application already making use of HorizSliderPane. The example usage was a football analysis program, in which five sliders are used to manipulate defensive player characteristics (e.g., speed, age, height), and the predicted consequences of the characteristics (e.g., sacks, interceptions, tackles) are graphed in a separate pane. This application uses five main classes (FootballAnalyst, HorizSliderPane, BarGraphPane, AnalysisWindow, and Dictionary; only Dictionary is part of the standard library; see Figure 1); method count ranges from five to 33, with from 118 to 363 inherited methods. Because one of our research goals was to examine experts' strategies for discovering and employing example usage information, the programmers were not told of the example application in advance.



```
Object
  Collection
    IndexedCollection
      OrderedCollection
        Network
          Hierarchy
            Library
            OrgChart
  NetworkConnection
  NetworkNode
  Window
    ApplicationWindow
      LibraryWindow
      NetworkWindow
        HierarchyWindow
          OrgChartWindow
    SubPane
      GraphPane
        NetworkPane
          HierarchyPane
      ListPane
```

```
ComputerApplications
  Engineering
    Electrical
    Aerospace
    Civil
  Humanities
    Social
    Behavioral
```

Figure 2. The Library Project: On the left is a listing of the major classes involved in the library and organization chart applications; indentation in the list signifies superclass-subclass relationships. In the upper right is the original library application; beneath it is the application enhanced to use a graphical hierarchy.

2.2 The Library Project

The second project consisted of enhancements to a library acquisitions application. This application manages a hierarchical collection of book categories (e.g., Computer Applications broken into Electrical Engineering, Aerospace Engineering, etc.); categories are annotated with information about acquisitions (e.g., number of books, titles). Hierarchical structure is conveyed via an indented list (see Figure 2), and users manipulate the categories by selecting a list item and making menu selections. In this way, they can add and delete categories, rename categories, and browse and edit the acquisitions information.

The library project uses five main classes (Library, NetworkNode, ListPane, NetworkConnection and LibraryWindow; only ListPane is part of the standard hierarchy, and the Library class inherits from two novel superclasses, Network and Hierarchy; see Figure 2). The method count for these five classes ranges from 4 to 54, with from 118 to 319 inherited methods.

Programmers were asked to enhance this project by using the new class HierarchyPane; again, they were told nothing of the appearance or functionality of the target class. An instance of this class is able to graph a hierarchical network of nodes (see Figure 2). It also can identify nodes or connections selected via a mouse click. Finally, the subpane allows users to name nodes by typing directly onto the graphed elements.

HierarchyPane differs from HorizSliderPane, in that much of its functionality is inherited from its superclass NetworkPane. Further, it was designed to work in concert with a number of other novel classes (HierarchyWindow, Node, NetworkNode and NetworkConnection), whereas HorizSliderPane is a relatively "standalone" component. A typical solution for reusing HierarchyPane in the library application involves creation of a new LibraryWindow class as a subclass of HierarchyWindow (thereby inheriting the ability to draw, select, and name nodes in the graph), and the updating of five methods from the original LibraryWindow class (the methods for adding, removing and showing acquisitions for a selected category, the method defining the menu, and the `openOn:` method).

As for the color-mixer project, the class hierarchy included an example usage of HierarchyPane — an organization chart, in which the nodes correspond to employees, and in which employees of various job descriptions (e.g., staff member, secretary, visitor) can be added to the hierarchy, given names, reassigned, and given project descriptions. The example uses seven main classes (OrgChart, OrgChartWindow, HierarchyPane, Node NetworkNode, NetworkConnection, and TextField; none of these are part of the standard hierarchy, and both OrgChartWindow and HierarchyPane inherit from novel superclasses; see Figure 2). The method count for these classes ranges from 1 to 37, with inherited methods ranging from 118 to 442. Programmers were not told in advance about the HierarchyPane usage example.

3 Reuse of Uses

In most discussions of component reuse in object-oriented systems, the focus has been on the class or classes reused. Design methodologies attempt to articulate characteristics of reusable classes [15,18] and tool builders develop techniques for classifying and retrieving useful classes [12,21] The dominant metaphor is "construction" — the programmer finds parts that can be reused, modifies them as necessary and connects them together (see, e.g., [4,5]).

Our observations suggest that this focus on components may be over-simplified. To develop the knowledge needed to reuse the components directly, the programmers would have had to stop work on their overarching goal — enhancing the project they had been given — and spend time analyzing and reflecting on the target class. These programmers were too focussed on their end goal to engage in protracted analysis. Instead, they made active use of all resources available in the environment, and began programming immediately. This led them to reuse the components only indirectly, through the reuse of "uses". That is, the main entity participating in the reuse programming was not the target class but rather the *example application of that class*. The programming consisted of finding and reusing the patterns of component reuse reified in the example application. As one programmer put it, on discovering the example application, "so there's a solution in the system!"

The extensive reuse of the example occurred despite mixed feelings expressed by the programmers. There was a sense that this wasn't the "right" way to reuse a class, that it was somehow cheating or taking the easy way out. One programmer said that he would look at the example only if all else failed, but then immediately began to work with it. Another viewed the example as a mixed blessing, because although it offered information on how to use the target class, it now required analysis itself: "Whenever you provide help, you provide trouble, now I have to understand this!" However, when probed about these feelings at the close of the experiment, the programmers indicated that the strategy of borrowing heavily from examples is one they use frequently in prototyping Smalltalk applications, and that their reservations were due to a perceived demand to use more conservative methods in this experimental situation.

Smalltalk provides explicit support for the identification and reuse of example usage context through its "senders" query which returns a list of methods in which a target message is sent. An experienced programmer can browse this list and make reasonable guesses as to which other classes if any are already using the class of interest; if motivated, they can then explore these other classes to discover why and how the target class is being used. All of the programmers made early and repeated use of the senders query; further, they showed an ability to discriminate among the various messages defined for the target class, asking for senders only on the more important methods (e.g., a method providing the contents for the subpane): "AnalysisWindow seems to be figuring prominently as a sender of interesting messages".

3.1 Reusing Pieces of an Example

The most common reuse of the example applications consisted of borrowing code used as the interface to the target class, both blocks of code copied out of methods and entire methods. For instance, all of the programmers borrowed code from the example applications' `openOn:` methods; by convention this is a message sent to a window which instantiates the various subpanes, defining their graphical and behavioral characteristics. The instantiation of subpanes in Smalltalk/V is often complex, and typically includes the definition of events that the subpane will handle. Thus copying an instantiation code snippet (8-15 lines of code) can save considerable time in working out exactly how a new kind of subpane needs to be initialized.

Sometimes the borrowed code was not directly reusable itself, but rather was used more as a functional specification. In working out slider event handling for the color-mixer project, the programmers copied over the `sliderActive:` method from the football program. This method does three things: first, the affected slider processes the mouse activity; second, the relevant player characteristic is updated; and third, predicted player performance is graphed. Only the first of these events maps directly to (and thus could be reused in) the color-mixer project. Nonetheless, the programmers were able to understand the code in `sliderActive:` as a specification of what they needed to do in their own version: process slider activity, re-set the model data (in this case, the color settings), and display the results (the new color swatch).

On a few occasions, the borrowed code came from work the programmers had just completed themselves (as in the "new code reuse" situation described in [6]). For example, both programmers working on the color-mixer first developed the code for one slider, then worked from that code to implement the other two. In these cases, the programmers knew exactly what needed to be changed, and the "programming" consisted simply of the physical edits.

In general, the copy/edit strategy worked quite well (see also [16]). It reduced the amount of typing required of the programmer, and helped to insure that the details of the code (e.g., placement of line separators) would be correct. More importantly, it removed the burden of analyzing the target class enough to generate the correct protocol for a particular usage situation, enabling a rapid programming progress. For many parts of the borrowed protocol (e.g., the event definitions in the `openOn:` method), the programmers knew what parts of the code needed to be edited and how to do this.

However, the copy/edit strategy did lead to some problems stemming from the novel parts of the target class' protocol, in that the programmers were now able to copy and "use" protocol that they didn't fully understand. A good example comes from one programmer's work on the color-mixer. In the football analyst example, each slider is instantiated with a different starting value. Because the slider instantiation code was copied from the football `openOn:`, instantiation of the `value` variable also became part of the color-mixer `openOn`. The `value` attribute is not generic to subpanes, so the programmer did not know off-hand whether it was prerequisite to slider functioning, and if so, what a reasonable starting value would be for the color-mixer. The programmer did not know enough about the protocol for sliders to answer these questions, so he simply made a guess. Later on, this guess caused problems, as the initial positions of the sliders did not match the starting color (white). Subsequently, the programmer solved the problem not by going back and correcting the initialization code, but rather by adding code at a later point that simulated the selection of white in the color list pane.

In some cases there was a conflict between the component interface suggested by the example, and the current design of the project. In the football program, the activity of all the sliders is handled by a single method `sliderActive:`. Modeling on the example, one of the programmers began by copying over the method and modifying it to refer to color-mixer objects. However, in the course of doing this, he recognized that there would be a problem in discriminating among the different slider instances. Despite the suggestion by the football example that multiple sliders could be managed by one method, he decided to change his approach and work from the more familiar model of the buttons used by the original user interface. Noting that three separate methods had been written to handle button activity, he developed an analogous set of three slider activity methods.

The Smalltalk environment is very supportive of the copying/editing of example usage code. Programmers can open as many code browsers as they like, and can freely select and paste text among them. In this study, the programmers almost always had at least two browsers open (one for the example and one for the project) and often used more when the code involved a number of embedded messages. In this way, they could preserve their top-level context while going off to answer a question or to find additional relevant code in other classes or methods.

3.2 Reusing an Application Framework

All of the programmers' initial efforts to reuse the example application involved bringing methods or pieces of methods *from* the example application *into* the project. However, the two programmers working on the library project ultimately decided to create a new kind of library window, one that was a sibling of OrgChartWindow (i.e., had HierarchyWindow as a superclass, this was in fact the solution requiring least programming effort). In doing this, they were deciding to *inherit* rather than borrow from the example usage context. After this decision, their activity shifted, as they began bringing code from the original library window into the new window. This was in marked contrast to the programmers working on the color-mixer project, who appeared to never even consider inheriting functionality from the football example.

The decision to subclass reflects a desire to reuse more than just the snippets of code involving the target class; in this case, the programmers elected to adopt the entire application context of the example. In Smalltalk/V PM, this context is normally managed by a window; the window communicates with the underlying application objects (e.g., a hierarchical collection of employees) and with the subpanes used to display application information. Thus reuse of the context can be accomplished by subclassing the application window; reuse of this sort is often referred to as reuse of an "application framework" [7]. Framework reuse brings along the component of interest "for free" in some sense, in that it is already a component of the framework, and the example window already has the code needed to interface between the component and other application objects.

Deciding to reuse the example's application framework had a remarkable effect on the programmers' reuse efforts. What had at first been a rather complex process of tracking down individual methods and instance variables distributed across NetworkWindow, HierarchyWindow and OrgChartWindow, and copying and editing methods or pieces of methods, now became a straightforward process of copying over and updating the menu functions from the original LibraryWindow class. One of the programmers spent over an hour reaching the decision to subclass; once he did, he was rather frustrated at the thought of throwing away all the work he had done so far, but even so was able to complete the project in fifteen minutes.

The problems of tracking down functionality distributed throughout an inheritance hierarchy have been noted before; Taenzer, Ganti and Podar [23] refer to this as the "yoyo" problem. The Smalltalk/V class hierarchy browser offers little support for dealing with hierarchically distributed function, as programmers must navigate from superclass to superclass in search of methods. Taenzer et al. [23] point to this problem as an argument against reuse via inheritance, suggesting that understanding how to subclass an extensive hierarchy requires much more distributed code analysis than simply reusing a component. Our situation offers a new twist on considering whether to reuse functionality directly or through inheritance: when a component has already been incorporated into a rich application framework, programmers may find that indirect inheritance of the component's functionality (i.e., through subclassing the framework) will simplify enormously the task of reusing the component.

Several factors seemed to contribute to the programmers' decision to reuse the application framework for graphical hierarchies. One was simply the difficulties in tracking down, borrowing and integrating function. There seemed to be a sense that the process was more complicated than it should be, e.g., "I should probably be trying to inherit some of this...". When asked later, one of the programmers indicated that it was his realization of how many of his borrowed methods were inherited from superclasses of OrgChartWindow that made him decide to move the library window. For the other programmer, a critical incident was his effort to compile a key method (the one allowing selection of nodes in the graph), and discovering a instance variable of the example window that had no analog in the library application. Up to that point, he had seemed willing to work with the complexity of tracking down and borrowing example protocol, but adding a new (and mysterious) piece of state information was too much.

Another factor may have been the similarity between the example usage and the project. On first discovering the HierarchyWindow class, one programmer tried a simple experiment while voicing his belief that it would never work: he tried opening a HierarchyWindow "on" the library object (an instance of Library, part of the Collection hierarchy). To his (and our!) surprise, this experiment was successful. Of course, the LibraryWindow functionality was not present, but at least the book collection was displayed in a nice graphical hierarchy. This experiment may seem extreme, in that it has a rather low probability of pay-off. However, it was simple to do, and it provided the programmer with considerable insight into the example application that he was able to apply to his later efforts.

The subclassing strategy did simplify the reuse programming project. However, it also introduced some rather subtle problems. There was considerable overlap in the functionality of the example and of the library (e.g., both had facilities for adding and removing elements in a hierarchy, for renaming these elements). One of the programmers, having decided to subclass, wanted to inherit as much functionality as he could. So, when updating the menu selections, rather than copying over the methods from the original library window and editing them to work in this new context, he first tried simply inheriting the methods defined in the superclasses. On the surface, this strategy seemed to work — he was now able to add and delete library categories and rename them. He never realized that the underlying library structure was not being manipulated correctly (the relationships among categories weren't being specified). It may be that programmers following a subclassing strategy are more likely to satisfice, accepting generic inherited functionality that is almost but not quite right simply because it is there and is already working.

4 The Reuse Programming Process

The programmers were opportunistic in the objects of reuse — extensive recruitment of the example contexts reduced considerably the amount they needed to learn about the target class. But they were also opportunistic in how they went about doing the reuse task. They spent little time in deliberated analysis of the example, in understanding how it was going to help or interfere with their enhancement efforts. Rather they began using the code of the example immediately to make progress on their goal. These were *active users* of Smalltalk [1]: as has often been observed for human problem-solving [11[the process we observed was very locally driven, with specific features of the environment and the evolving solution determining each succeeding step.

4.1 Getting Something to Work With

An early goal for all of the programmers was to get an instance of the target class up and running, so that they could see what it looked like. One of the programmers working on the library project was able to use the organization chart example to do this. After discovering the example, he immediately took on the goal of starting it up. He found an OrgChart class method `fromUIIData`, the name of which signalled to him that it was a special "set-up" method, and that he could use it to create an appropriate OrgChart object and start up the application. By doing this, he was able to see what a HierarchyPane looked like, as well as to experiment with the interaction techniques it supported.

With respect to programming activities, the focus of initial efforts for all of the programmers was on modifying the project's `openOn:` method to include the new class: "I want to get one of these things as a subpane". However, while there was some browsing of the target class methods to see how to do this, the browsing tended to yield inferences about class functionality rather than usage protocol; as we noted earlier, the programmers seemed to resist carrying out an analysis of the target class comprehensive enough to allow them to write code to instantiate it for their project. Instead, they sometimes looked for clues in the code they were replacing. Thus the two programmers working on the color-mixer examined the code used to create the buttons, thinking about how they might modify it for sliders (e.g., what events a slider might handle in contrast to a button).

One programmer working on the color-mixer tried to take advantage of other code in the `openOn:` method as well. Noting that HorizSliderPane is a subclass of GraphPane, he examined the code instantiating the color swatch (an instance of GraphPane), thinking that he might be able to build a slider definition from it. This led to a variety of problems, as he began to hypothesize that the slider functionality was somehow built from the scroll bars present in every subpane, and that the protocol controlling these scroll bars for GraphPanes must be critical in creating sliders. This was certainly a reasonable hypothesis on functional grounds, but in fact was quite misleading.

The programmers seemed to feel that successfully instantiating the target class within the project context was a momentous event. It appeared that this was considered to be the major hurdle of the project, and now they could get on with business as usual, adding the remainder of the component's functionality (i.e., its event handling). One explanation for this is that the programmers could "see their end goalin sight" — a new and improved view of their project data. But another equally important factor is that by instantiating the new component as part of the project, the programmers could now rely much more on the environment to guide their programming. In a Smalltalk application, objects are created and code references are established only when the application is run, making the code alone inherently ambiguous and mental simulation of it difficult. In contrast, if the programmer is able to start up an application, all ambiguities in the code are resolved, and the programmer can use Smalltalk's sophisticated interactive debugging tools to analyze and modify the code.

4.2 Debugging into Existence

We have seen that the programmers relied heavily on code already in the environment in attacking the reuse projects. But they also relied heavily on the tools of the environment to locate and make sense of the relevant code. In particular, they repeatedly started up the application they were working on, and looked to see where it "broke" to plan their next move.

Smalltalk is particularly supportive of this debugging-centered style of program construction. The language is non-typed and compiled incrementally, which permits rapid and repeated experimentation with the code used to run an application. The debugger and inspector tools support such experimentation directly, providing flexible access to and manipulation of the runtime context for an application (i.e., objects and their state, messages in progress).

In some cases, the programmers knew something of the steps they would need to take, but used the debugger to help them in carrying these out. Thus, once they had copied the instantiation code from the example application's `openOn:`, they knew that certain modifications would be necessary: instance variable names needed to be changed, the menu name needed to be changed, the project would need a drawing method, etc. Some of the programmers even carried out some anticipatory activity, perhaps creating a method that they knew they would need, but that they also knew was not yet functional. However, for the most part, they relied on the system to detect the absence of methods or the inappropriate states of objects. In a typical scenario, the programmer would start up the project application, receive a "message not understood" error, return to the example in search of a method with that name, copy the method, perhaps making a few changes, try again and see how far it got, make some changes and try again. This sort of cycle might be repeated many times, but the programmers seemed comfortable with it, and seemed confident that they were making progress.

In other cases, the debugger was used to untangle more subtle problems. So, for example, the superclass HierarchyWindow uses the `network` instance variable to point to the main application object, whereas the original LibraryWindow class uses `library`. A thorough analysis of the example would have revealed the relevant mapping between these two variables. However, the two programmers working on this project simply borrowed the example code as-is and used the debugger to ascertain what role the `network` variable was playing and how to provide this information within their project.

The compiler was used in this opportunistic fashion as well. When dealing with complex pieces of borrowed code, the programmers often would attempt to compile the code before they had completed editing it. The system would flag variable names not defined for the class (e.g., the HierarchyWindow code refers to `graphPane`, while the LibraryWindow uses `pane`), and the programmers would then replace the unknown name with the name of the analogous variable. This minimized the amount to which they needed to read through and analyze the unfamiliar code.

5 Summary and Implications

Our observations describe a process of component reuse in which the component is reused only indirectly, through the reuse of its "uses" — bits of protocol or even entire application frameworks. The programmers we studied pursued this style of reuse piecemeal and opportunistically; they focused initially on getting a runnable albeit skeletal result which they could exercise and improve incrementally, relying heavily on interactive debugging. We have characterized these as "active" programming strategies, an orientation in which programmers directly and immediately enlist and transform their software materials in favor of withdrawing from such activity to analyze and plan.

5.1 Scope of Active Reuse

This work was exploratory empirical research in its scope and scale. It addressed a particular programming situation, application prototyping, which may differ significantly from other situations. However, at least some of our observations are consistent with studies of other reuse situations. Lange and Moher [16] observed that an experienced programmer extending a library of software components was quite likely to use existing components with related functionality as templates or models for the new components. Detienne [6] found that programmers designing and implementing new applications somtimes reused their own code as they worked. Interestingly, the programmers in this study chose not to borrow code from other applications, perhaps because the other applications available were only peripherally related to the problems being solved.

Further research is needed to assess the generality of the more specific strategies we observed. All four programmers relied extensively on the system tools to organize their work, using multiple browsers to maintain their context across different parts of the hierarchy, and using the debugger and inspector to track down and modify missing or inappropriate pieces of borrowed code. It is not clear though what the boundary conditions for such an approach might be — it may be that they are only likely to occur in a tool-rich interpreted environment like Smalltalk.

Some strategies were unique to a particular programmer. For example, only one programmer made the effort to "run" the example application before borrowing code from it. He felt that this gave him a chance to preview the functionality he would be incorporating; it may be that across a wider variety of reuse projects, perhaps involving more complex components, such a strategy would be more prevalent. In another case, one programmer experimented with opening a graphical HierarchyWindow "on" his application data. The success of this experiment conveyed a great deal to him about what the graphical network framework expected in terms of data structures. It is important to understand the generality of such techniques and strategies.

5.2 Consequences of Active Reuse

Beyond the question of generality, we can ask about the consequences of the active programming strategies we observed. For example, two of the programmers did not produce a perfectly correct result, and it is not clear whether or how their problems would have been detected and corrected given unlimited time, or given instructions that emphasized the accuracy of the result. Indeed, the active programming we observed may be inadvisable from a software engineering perspective, if the small errors or inefficiencies introduced by reliance on example code are very difficult to unearth subsequently. Further research is needed to determine what if any strategies experts have developed for minimizing this downside inherent in reuse by example.

It is important not to lose sight of the main benefit of this style of software reuse: these active strategies reflect a creative and effective resolution of the inherent tension between the need to distance oneself from one's own project to study someone else's code, and continuing to make concrete progress toward a desired result. Elsewhere we have characterized such a tension as the "production paradox" [1], wherein users are too focussed on the product they are creating to acquire the skills that will facilitate its creation. In this Smalltalk reuse setting, the programmers' borrowing of example code allowed them to quickly incorporate at least some approximation of the new functionality into their own project; they could then work within their own project context to "learn" the minimum necessary for successful reuse.

5.3 Training and Tools for Active Reuse

Our work has a variety of implications for how objected-oriented programming should be conceptualized, taught and supported. Most generally, it suggests the desirability of a broader view of component reuse: the pluggable "software IC" metaphor [4,5] is not the only way reuse has been conceptualized, but it is a dominant image in talking and thinking about reuse. Both of our target objects (the slider and the graphical hierarchy) could be used as pluggable components; the slider, in particular, is an interface widget and eminently pluggable. However, all four programming projects described here reused the target classes through use of some or all of their example usage contexts. This suggests a more situational view of reuse in which pluggable, context-free reuse is the simple and ideal case.

The programmers we studied invented the strategies we observed or learned them informally from colleagues. As we noted, they occasionally expressed some embarrassment at their own reluctance to fully analyze code they wanted to reuse and their predilection for "stealing" usage protocol. If these practices survive — indeed emerge from — the natural selection pressures of professional programming, we should at least consider that perhaps they should be the topic of instruction in (Smalltalk) programming.

This implication for instruction entrains a related implication for the documentation of software components. Our four programmers were able to find example uses of the target classes, but in many situations this would not be true, and hence an example-oriented reuse strategy would be thwarted. Of course, imagining example-based documentation on a large scale raises many consequent issues. Who will build the examples? One resource is the test programs built in the course of development, and often discarded afterward. Delivering these along with software components would provide some support for the example-oriented strategy at virtually no cost. Another question is what makes a good example. There is a literature on concept formation in cognitive psychology that addresses the issue of how examples are abstracted in comprehension [19]. It is an interesting and open question whether and how similar characteristics bear on reuse.

Finally, this work embodies three themes for tool support: the sequence of activities in reuse programming, recruitment of example usage code, and the use of the system debugger. Our four programmers seemed to follow a loose script: first they instantiated the component in the project context, then they successively elaborated it function by function. Throughout this process they made extensive use of example usage contexts and of the debugger. An obvious implication is to provide tools that more explicitly integrate and coordinate the information needed at each point along the way. Thus tool support might guide reuse activities through a reuse script (for example, a list of target class behaviors to instantiate in the project context), using this script to coordinate the programmer's work with the example usage code, the project code, and the interactive debugging facilities.

Acknowledgements

Important contributions to this work were made by Christine Sweeney and Dan Walkowski, who assisted in the design and development of the reusable components and the example usage contexts.

References

1. J.M. Carroll and M.B. Rosson. The paradox of the active user. In J.M.Carroll (Ed.), *Interfacing thought: Cognitive aspects of human-computer interaction* (pp. 80-111). Cambridge, Mass: MIT Press, 1987.

2. J.M. Carroll and M.B. Rosson. Deliberated evolution: Stalking the View Matcher in design space. *Human-Computer Interaction, 6*, 281-318, 1991.

3. J.M. Carroll, J.A. Singer, R.K.E. Bellamy, and S.R. Alpert. A View Matcher for learning Smalltalk. In *CHI'90 Proceedings* (pp.431-438), New York: ACM, 1990.

4. B.J. Cox. *Object oriented programming: An evolutionary approach.* Reading, Mass.: Addison-Wesley, 1986.

5. B.J. Cox. Building malleable systems from software 'chips'. *Computerworld* (March), 59-68, 1987.

6. F. Detienne. Reasoning from a schema and from an analog in software code reuse. In J.Koenmann-Belliveau, T.G.Moher & S.P.Robertson (Eds.), *Empirical studies of programmers: Fourth workshop.* (pp.5-22). Norwood, NJ: Ablex, 1991.

7. L.P. Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In T.J.Biggerstaff & A.J.Perlis (Eds.), *Software reusability, volume 2: Applications and experience* (pp. 57-72). New York: Addison-Wesley, 1989.

8. Digitalk, Inc. (1989). *Smalltalk/V PM.* Los Angeles: Digitalk, Inc.

9. K.A. Ericsson and H.A. Simon. Verbal reports as data. *Psychological Review, 87,* 215-251, 1980.

10. G. Fischer. Cognitive view of reuse and redesign. *IEEE Software (July)*, 60-72, 1987.

11. B. Hayes-Roth and F.A. Hayes-Roth. A cognitive model of planning. *Cognitive Psychology, 3*, 275-310, 1979.

12. R. Helm and Y.S. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. *Proceedings of OOPSLA'91* (pp. 47-61). New York: ACM, 1991.

13. S. Henninger. (1991). Retrieving software objects in an example-based programming environment. *Proceedings of SIGIR'91* (pp. 251-260). New York: ACM, 1991.

14. R.E. Johnson. Documenting frameworks using patterns. In *Proceedings of OOPSLA'92* (pp. 63-76). New York: ACM, 1992.

15. R.E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-oriented Programming 1(2)*: 22-35, 1988.

16. B.M. Lange and T.G. Moher. Some strategies for reuse in an object-oriented programming environment. In *Proceedings CHI'89* (pp. 69-74), New York: ACM, 1989.

17. J.A. Lewis, S.M. Henry, D.G. Kafura, and R.S. Schulman. An empirical study of the object-oriented paradigm and software reuse. In *Proceedings of OOPSLA'91* (pp. 184-196). New York: ACM, 1991.

18. B. Meyer. *Object-oriented software construction.* New York: Prentice Hall, 1988.

19. R. Millward. Models of concept formation. In R.E.Snow, P.-A. Federico & Montague, W.E. (Eds.), *Aptitude, learning and instruction: Cognitive process analyses.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1979.

20. L. Neal. A system for example-based programming. *Proceedings of CHI'89* (pp. 63-68). New York: ACM, 1989.

21. R.K. Raj and H.M. Levy. A compositional model of software reuse. In *Proceedings of ECOOP'89* (pp. 3-24), London: British Computer Society, 1989,

22. M.B. Rosson, J.M. Carroll, and C. Sweeney. A View Matcher for reusing Smalltalk classes. *Proceedings of CHI'91* (pp. 277-284). New York: ACM, 1991.

23. D. Taenzer, M. Ganti, and S. Podar. Problems in object-oriented software reuse. *Proceedings of ECOOP'89* (pp. 25-38). Cambridge: Cambridge University Press, 1989.