In Proceedings of the Western Computer Graphics Symposium 2000, (Panorama Mountain Village, BC, Canada), March 26-29, 2000.

Change Management

James Tam, Saul Greenberg, and Frank Maurer
Department of Computer Science
University of Calgary
Calgary, Alberta
phone: +1 403 220 3532

tamj@cpsc.ucalgary.ca

ABSTRACT

In this paper, we analyze the problems of keeping up with diagrammatic changes being made within a collaborative software design tool. With these graphical tools, one software engineer may specify possible software architectures e.g., through a UML editor. A second software engineer may then modify the diagram. The problem is how the original engineer can track what changes had been made. Most systems provide little or no support for this, and we believe that relying on the engineer's memory is inadequate. We propose a several graphical representations that can illustrate to an author what has changed. Issues in graphical representation of changes include how actual changes can be portrayed, as well as ways to filter the view to ensure the designers only see relevant changes at an appropriate level of detail.

Keywords: Change Management, information filtering, graphical representations, graphical software design tools.

1. INTRODUCTION

Teams involved in collaborative software design and engineering often follow a divide and conquer approach. They try to split the project into manageable tasks, where a single person can work on a task. The challenge is in bringing the deliverables together into a cohesive and testable whole.

However, there are also times when team members must work on the same task and task artifacts: documents, software, UML diagrams, functional specifications, and so on. One person may work on the artifact for a while, and then pass it on to the next person (perhaps to continue the work or to revise it as required). Iterative development can then occur between these two people in a back and forth manner, or it could even include other team members.

The problem is how team members can recognize and / or track changes made in a document by other members. At the crudest level, this could simply be a case where one person analyzes a changed artifact, perhaps relying on memory to recognize changes. Errors and inaccuracies are, of course, likely. Slightly better is the case where people communicate over the document, where one tells the other (perhaps orally or by notes within the document) what has been done. Of course, this requires a great deal of additional work, and it is easy for one person to neglect to tell another about some crucial change.

What is needed is a more formal way to support how people recognize changes within artifacts. We thus define *change management* as a process that helps one person recognize and track changes made by one or more others.

Our particular interest is in how change management can be applied within a diagramming tool, such as a UML editor. We are especially concerned with how changes within these tools can be tracked, how the relevancy of changes can be determined, and how changes can be displayed to another person in an effective and efficient manner.

We set the scene by first describing previous work on change management in both human computer interaction and in software engineering. We then discuss the difficulties of doing change management in a graphical diagramming tool, especially when many changes require some notion of filtering to reduce complexity. Finally we will present and discuss our early work in how to represent changes within a graphical editor.

2. PREVIOUS RESEARCH

2.1 Software Engineering research in change management

Within the field of Software Engineering, researchers have stated that the goal of change management is to be able to predict how a software project will be affected by the changes that are made to the project [1]. This is somewhat different from our view of change management: while we are concerned with *what* changes have been made, this other view considers the *effect* of changes.

Still there is work related to our own definition of change management. In particular, almost all programming environments contain some kind of version control system. One example is the CVS (Concurrent Version System) available in Unix, and Microsoft Visual SourceSafe available as part of the Microsoft Visual Development Suite. Both allow programmers to check in and check out versions of programs and other documents as they are being developed, and allow comments to be added to them (such as a textual note describing what has changed). They also allow differencing of versions, where differences in text sequences are shown.

Dellen [2] developed a different type of change management framework that would automatically notify the interested members of a team when particular changes occurred. It used an event-driven notification system: as a change occurred in a piece of software, an event would be raised. If developers within the project had registered their interest in that particular type of event,

then they would be notified of the change. Similar to this, programmers hooked the Elvin notification server [3] into CVS, where notifications of how files were checked into and out of the repository appeared on a one line tickertape. As with Dellen's system, programmers could subscribe to those items they were interested in.

While these strategies are interesting, all are somewhat difficult to apply to the graphical nature of diagramming systems. Programmers may find it difficult to articulate a graphical change: in real life, we often gesture around the drawing to do this. This makes the current notification and comment annotation components of systems somewhat unwieldy. The version differencing tools only work on sequential text: they are not able to show differences within a graphical drawing, such as would be found in a graphical design editor for the Unified Modeling Language (UML).

Unlike these other systems, which are centered on sequential text, Rational Rose (by Rational Software Corporation) contains a change management facility that works in a UML diagramming tool. It works by translating the diagram into a hierarchical text description and by highlighting changed items within this text. Unfortunately, this representation of the UML diagram and its changes are no longer in graphical form: thus programmers must view it in a different (and perhaps more difficult to understand) representation. As well, an approach such as this would not be able to handle free form annotations and marks that can be added to the UML view, as can be done in the Argo open source UML editor [4].

2.2 HCI research in change management

As with software engineering, most of the previous research has focused on text based work environments. Perhaps the best example is Neuwirth et. al.'s "Flexible Diff-ing" text differencing system [5] which was developed by Neuwirth, Chandhok, Kaufer, Erion, Morris, Miller. What makes it special is that it allows viewers to contrast changes at various levels of detail. Thus changes can be viewed at a high level (e.g., where have changes been made) as well as in progressive detail (e.g., exactly what changes have been made).

Hill and Hollan [6] proposed one graphical approach related to change management called "edit-wear and read-wear". They would track what parts of a document had been either read or edited, and would then use graphical "wear" indicators to indicate how much had been changed and in what places. The more often that a portion of text was changed, the more vivid the wear indicator [6]. While there was a brief discussion of how some of these ideas might be applied to a graphical based environment, such as applying wear indicators to user interfaces, the main focus of the research was conducted in text systems.

3. CHANGE MANAGEMENT ON COLLABORATIVE GRAPHICAL DOCUMENTS

Our particular interest is how change management can be supported in a collaborative process that uses predominantly graphical rather than textual documents. While we are interested in how change can be tracked within all 2-dimensional graphical drawing and diagramming applications, we will concentrate for now on how software engineers collaboratively develop UML diagrams. Our approach is to somehow track and visualize changes within a diagram so that engineers can answer questions such as:

- Have any changes occurred since I last visited this document?
- How many changes have occurred?
- Where have these changes happened?
- How have particular parts of the diagram changed?
- Who did these changes?
- Why did they perform these changes?

These questions were derived from similar questions raised by Gutwin [7], who was studying how people would track what others were doing when working together in real time over a visual work surface. Gutwin was interested in what he called Workspace Awareness . While related, our own work will focus on awareness of changes in an asynchronous visual work surface designed for software development rather than a real-time type of system.

In the following section, we raise two issues that we believe must be addressed by any graphical change management system: information filtering and the techniques used to represent changes visually.

4. INFORMATION FILTERING

We expect some graphical documents to change little between versions, and some to change quite a bit. We expect cases where changes pervade the entire document, and others where they are quite localized. The problem is that in all these different cases, the viewer of the document must somehow make sense of what has changed. Showing all changes at all levels of detail may be confusing when many changes are present, and people will have to do much work in order to determine which of the many changes are relevant.

One solution is to apply information filtering techniques to change management. This involves having the system somehow screen all changes that have occurred, and showing only the important changes to designers. Particular changes may be shown at a higher level of abstraction i.e., the abstraction could indicate that an object has changed, without detailing all the changes within it. Without filtering, designers may become bombarded with volumes of changes that they may or may not always be interested in.

4.1 Filtering: Too much vs. not enough

An important issue that is immediately raised is determining which changes should be shown to designers and which ones should be hidden. If too many changes are filtered then there is a risk that important information may be lost. If too few changes are filtered then the person may be overloaded with irrelevant information.

There are two approaches for determining the relevancy of a change to a designer. The first way is to try to automate this process and having a program determine what is relevant for an

individual. The second way is to allow people to decide the issue for themselves.

As previously mentioned, there was already some research conducted using the first approach by Dellen [2]. Her approach was tailored for situations where programmers were working on different but inter-related parts of a system, which differs from our situation where people are working on the same part of the system (or diagram).

In the second approach, the people would set their own criteria for determining relevancy. Only the changes that have meet these criteria would be displayed. For instance, a person may only be interested in seeing changes that occurred during a certain period of time, or seeing changes that were caused by a certain person. The problem, of course, is in giving people appropriate ways to determine relevancy within the tool.

5. GRAPHICAL REPRESENTATIONS

Assuming that a diagramming tool knew *what* changes are relevant to show to its user, it still must decide *how* to show these changes to the person. A key issue is the visual representation used. Any indicator of change must be noticeable enough so that it is easily interpreted and not overlooked, while remaining unobtrusive so that it does not interfere with the real work of software design.

The first step to finding good representations is to determine the classes of changes that can be made to a diagram. Because we are still in the early stages of our work, we have explored only three primitive change operations that people can apply to a UML diagram: the addition, deletion, and modification of objects.

There are, of course, many possible ways to represent these operations to a viewer. We are beginning our work with simple change indicators: icons attached to objects that indicate their changed state. Because there are many types of icons, we have developed and are testing the effectiveness of three different sets of change indicators.

1. Rudimentary graphical indicators use simple symbols to represent changes + for addition, - for deletion, and ▲ for modifications.
2. Change icons often seen in today's systems, with for addition (the blank document often represents 'new', for deletion, and for modification.
3. Text-based icons: Add for addition, Del for deletion, and Mod for modification.

In the following three illustrations, these representations will be shown in a sample software project specified as a UML class diagram. The changes added to the sample shows the situation where two classes have been added to the UML diagram, and one class has been modified by having a method deleted and a data field added to it.

6. FUTURE WORK

We are currently running a study to evaluate the effectiveness of iconic change indicators in general, as well as how particular change indicators perform. We are implementing and testing these simple change management ideas by modifying an existing UML editor. Of course, iconic change indicators are just scratching the surface of how to represent changes, and we expect to develop other much more radical methods for representing change as well as for filtering changes.

References

[1] Bohner, S. and Arnold, R. An Introduction To Software Change Impact Analysis. Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA 1996.

[2] Dellen, B. Change Impact Analysis Support For Software Development Processes. A Ph.D. Thesis from the University of Kaiserslautern 2000.

[3] Parsowith, S., Fitzpatrick, G., Kaplan, S. and Segall, B. (1998) Tickertape: Notification and Communication in a Single Line. In *Proceedings 3rd Asia Pacific Computer Human Interaction (APCHI98),* Japan, IEEE Computer Society, pp 139-144.

[4] Robbins S. An open source UML editor (v 0.5.2) from the University of California, Irving. Irving, California Available online at http://argouml.tigris.org

[5] Neuwirth, C., Chandhok, R., Kaufer, D., Erion, P., Morris, J. and Miller, D. (1992). Flexible Diff-ing In A Collaborative Writing System. Proceedings of the ACM '92 (Toronto ON, 1992) ACM Press, 147 - 154.

[6] Hill, W. and Hollan, J.. Edit Wear And Read Wear. Proceedings of CHI '92 (Monterey CA, 1992) ACM Press, 3 - 10.

[7] Gutwin, C. Workspace Awareness In Real-Time Distributed GroupWare. A Ph.D. Thesis from the University of Calgary, 1997.

1a. Change Indicators

Operation	*Change Indicator Used*
Addition	+
Deletion	-
Modification	▲

1b. Example

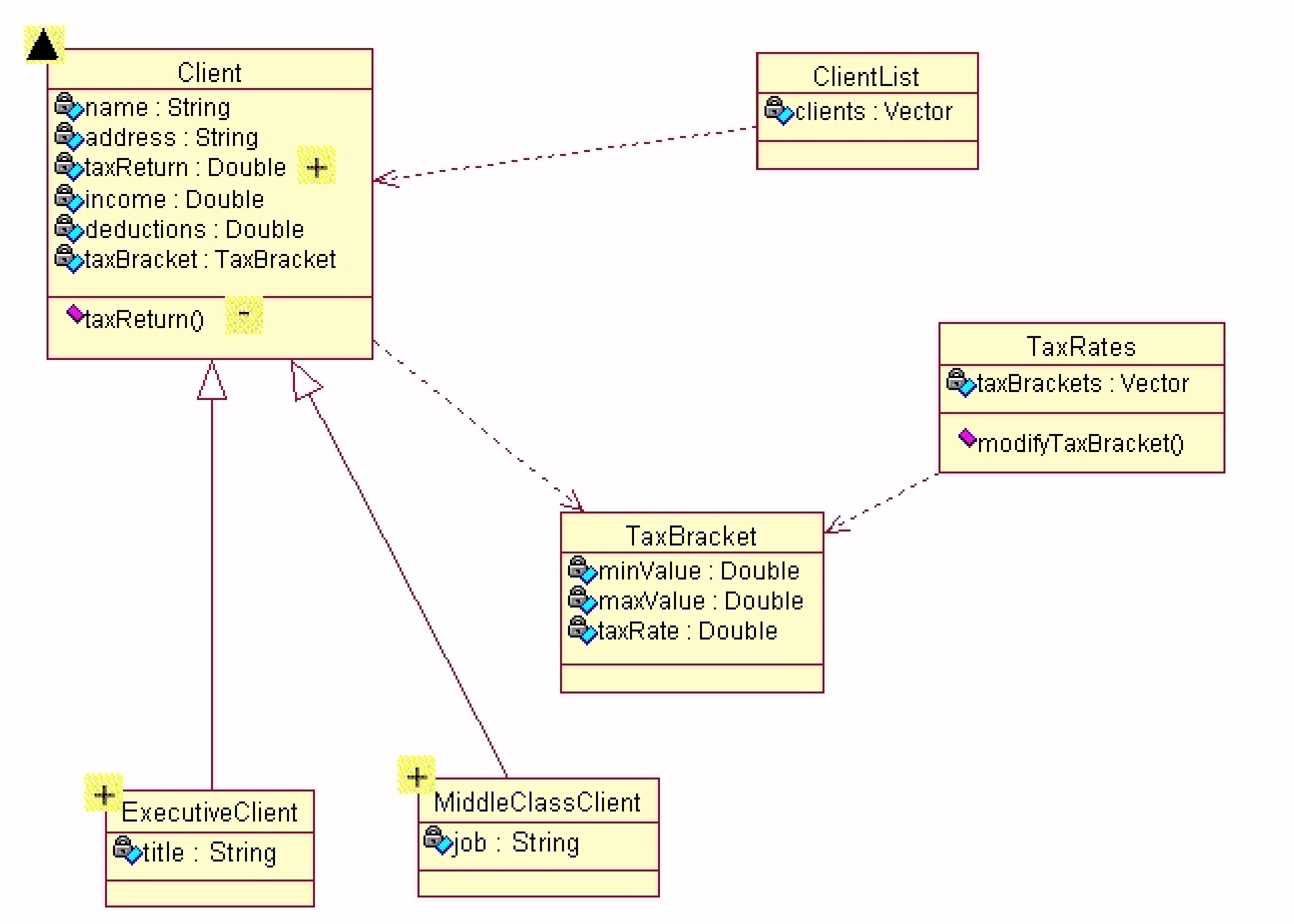
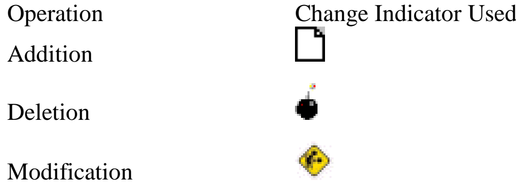


Figure 1. Rudimentary Graphical Indicators

2a. Change Indicators



Operation	Change Indicator Used
Addition	
Deletion	
Modification	

2b. Example

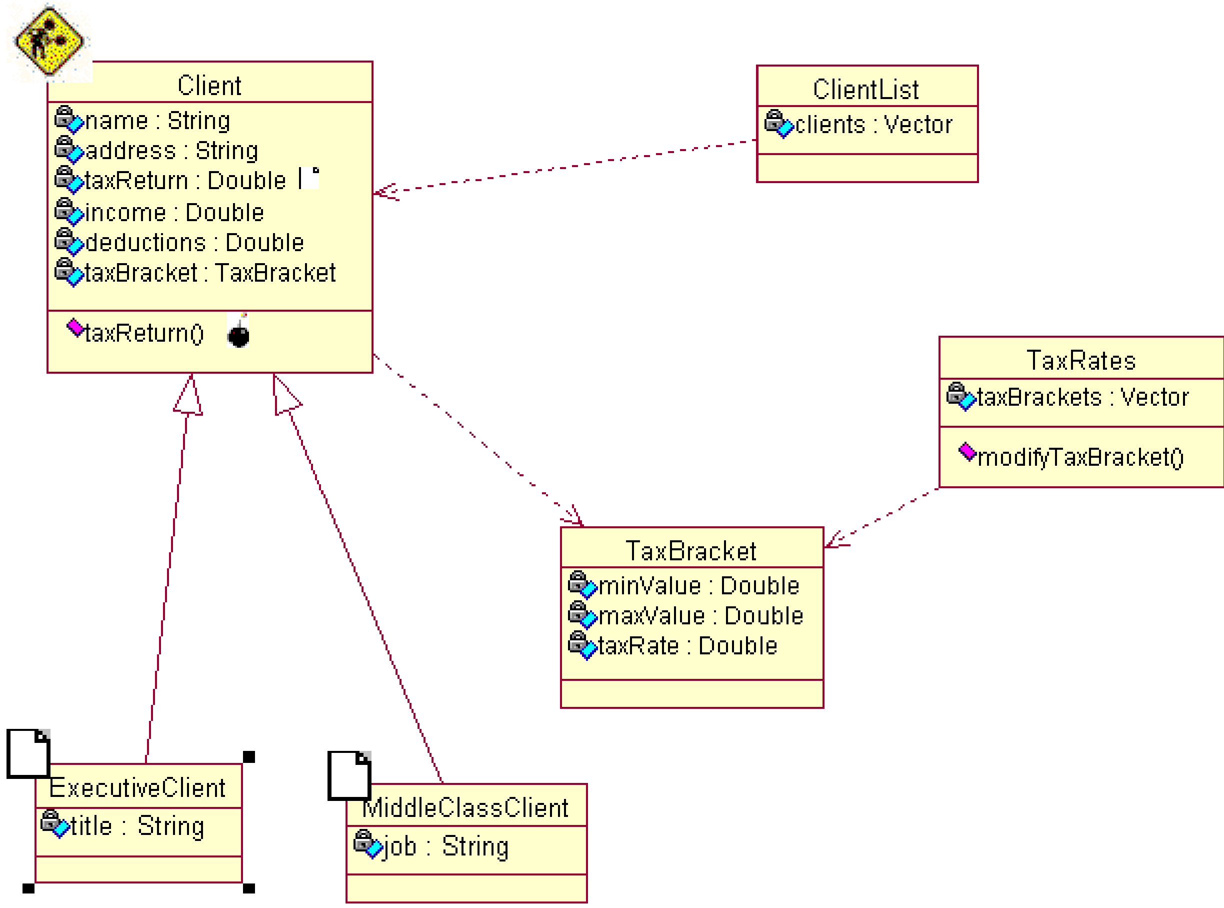


Figure 2. Common software based icons as change indicators

3a. Change Indicators

Operation	Change Indicator
Addition	Add
Deletion	Del
Modification	Mod

3b. Example

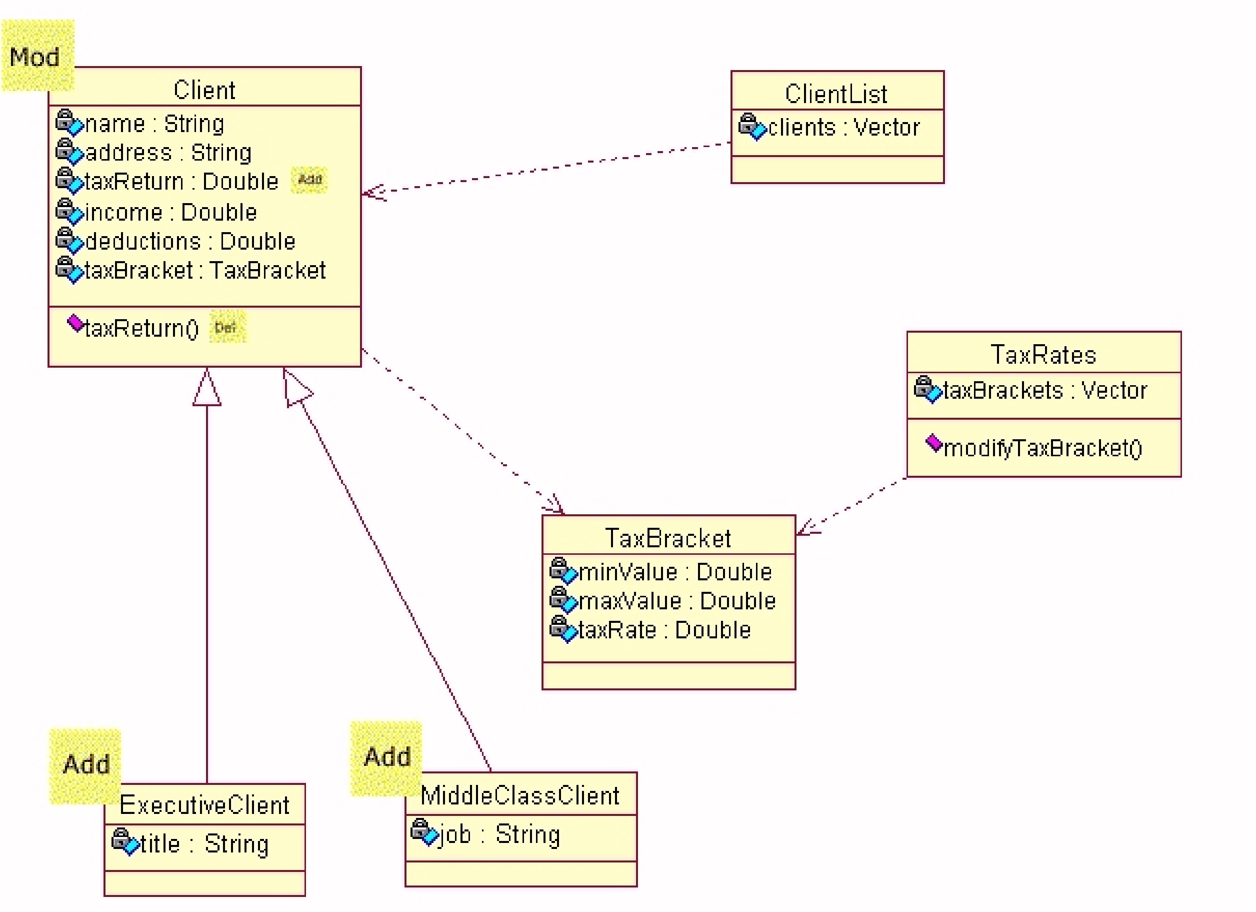


Figure 3. Text-based change indicators