Chapter 2
Previous research into change awareness

In this chapter, I describe existing approaches and system designs used to manage or display changes in collaborative projects. I start with early mechanisms for controlling and tracking changes between versions of text files and between collections of revision controlled files. Next, I survey popular techniques for displaying changes, as used by text file differencing systems built into both commercial and research word processing systems. These approaches work mostly with text. In the final section, I briefly present the few systems that support change in two-dimensional graphical systems.

2.1 Version control and configuration management systems

In this section, I describe a progression of standard systems that track change at a document or file level. Most work best with ASCII text documents (ASCII-based source for programs) and collections. I cover basic concepts including: manual and automatic maintenance of changes; conflict management; interdependencies between files and modules in a collaborative project; version and change granularity; providing a high level overview of all changes; and the management of binary files.

2.1.1 Manually tracking changes through backups

When people have no automated change support, they usually make backups of files as a manual way to save versions and track changes. Backups are useful both in individual and group projects: each time that a file is changed, another copy of the file would be made. This is done automatically for users of the VAX/VMS operating system (Compaq 1975). In both cases (manual and automatic), when someone wants to examine the changes made over time, they simply restore different backup files and visually compare older versions with current versions (Tichy 1991).

Not surprisingly, this technique has problems. In projects involving many changes by many people, backup versions accumulate so quickly that people have trouble managing the collection. They end up spending much of their time browsing and deleting useless files (Tichy 1991). As well, they often find it difficult to understand exactly what has changed between the many backup sets for there is no automatic way of distinguishing between versions containing trivial *vs*. important changes or of even finding the changes. Additionally, because manual backups requires that attention be diverted from the main task of making changes, people often neglect or forget to make them, even when their changes may have significant and widespread ramifications.

2.1.2 Version control systems

Version control systems automate and enhance the process of manually tracking progressive versions of documents and their changes. Version control is defined as the task of keeping the versions and configurations of a software system well organized (Tichy 1991). That is, these systems help people save and track different versions of individual documents (Magnusson and Asklund 1996). Example systems include the RCS Revision Control System (Tichy 1991), IBM's Clear/Caster system (Brown 1970), AT&T's SCCS Source Code Control System (Rochkind 1975), CMU's SDC Software Development Control System (Habermann 1979), and Digital Equipment Corporation's CMS Code Management System (DEC 1982).

All version control systems work by separating the editing of a document from the version control of a document. The first revision (version) is created from the document, which effectively "freezes" it. The frozen version can no longer be changed, and so editing it implicitly creates a new revision (Tichy 1991).

Version control systems overcome some of the problems found with the manual comparison of document versions by displaying differences through an algorithm such as *Diff* (Hunt and McIlroy 1975). Diff produces "…a small, line-based delta [a sequence of commands needed to transform one file to another] between pairs of text files" (Tichy

1991: pp. 7). Differences are determined on a line-by-line basis so that if only a single character in a line is changed, the program would flag the entire line as being changed. This approach will be discussed further in Section 2.2.

Version control systems also try to minimize conflicts that can occur when two or more people try to edit the same version. That is, the system controls the process of change to, "…detect conflicts and prevent overlapping changes" (Tichy 1991: pp. 1). This control is usually implemented through a "locking" mechanism that only allows one person to edit a version at a time (although this mechanism can be bypassed if so desired).

While useful for managing simple file sets, version control systems are limited. In an active project of moderate size that consists of interdependent parts, managing the changes of many authors remains daunting. Changes made in one part of the project can result in further, unexpected, changes in another part. It is hard to predict all of the possible consequences of a change made to complex systems without computer assistance (Luqi 1990; Arango, Schoen, Pettengill and Hoskins 1993). As a result *configuration management* systems were developed, which will be discussed in the next section.

2.1.3 Configuration management systems

Version control systems work best for simple file collections with few interdependencies and only a modest number of authors. Projects involving multiple authors and many interdependencies are better served by sophisticated configuration management systems, such as CVS (Berliner 1990), Visual SourceSafe (Microsoft 1992) or the IDE (integrated development environment) VisualAge (IBM 1991). As summarized by the British Standards Institute (1984), configuration management identifies the total configuration (i.e., the hardware, firmware, software, services and supplies) at any time in a system's life cycle, together with any changes or enhancements that are proposed or are in the course of being implemented. Thus, it allows changes to be traced through the lifecycle of a system or across a group of associated systems.

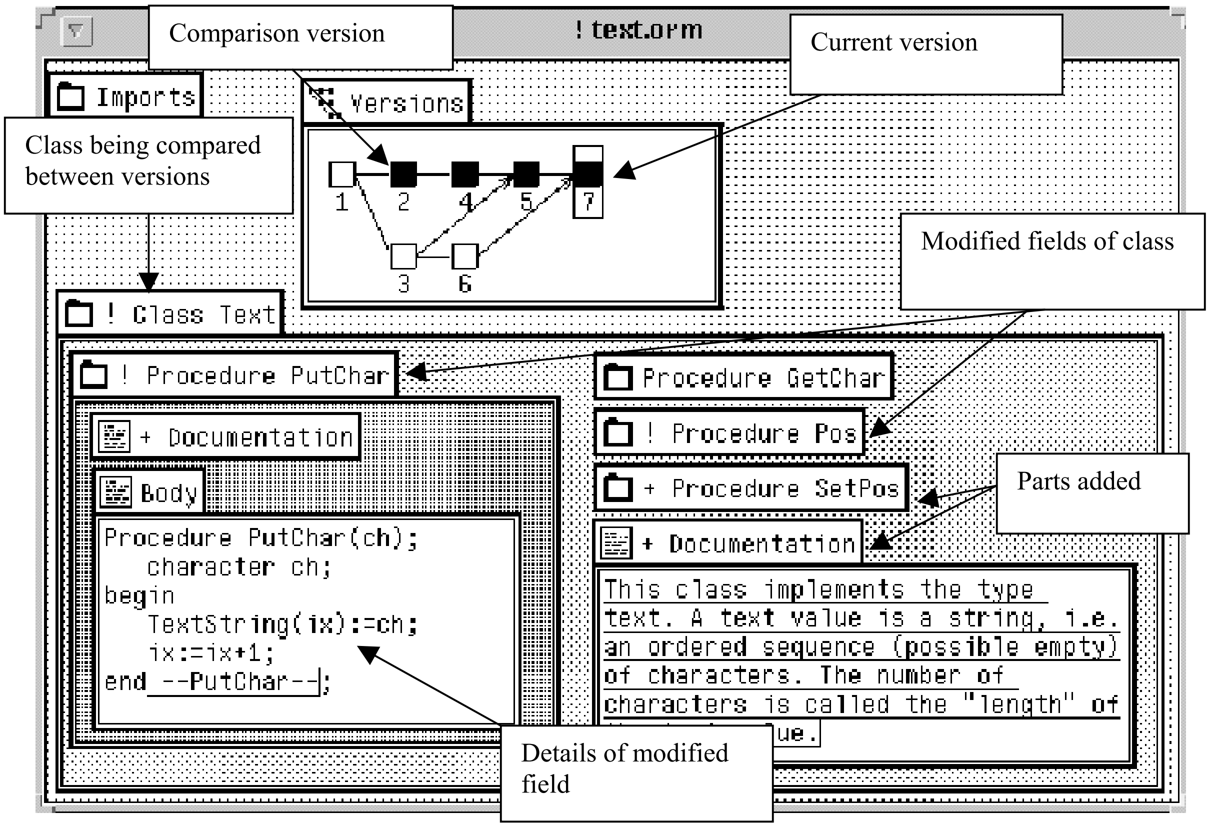
The main purpose of configuration management systems is to keep developers aware of the status of different software components at any point in time. Without awareness of all the components in a configuration, each modification made could result in disaster because the consequences of the change are unknown (Allan 1997).

2.1.4 Limitations of version control and configuration management systems

The first drawback of some version control and configuration management systems is that the number of changes between versions is quite large. Most version control and configuration management systems tend to be employed in cases where the number of changes across files is numerous and where the granularity of changes between files versions is quite coarse. As described by Sobell (1999) "…when you are fixing a collection of bugs in a file, you should fix each one and completely test it before recording the changes in a delta (version). This technique saves you from having deltas that reflect incomplete, transitional changes in the history of a file" (pp. 581).

When files contain well-defined parts, it may be more helpful to track changes of these parts as well as of the whole file. This is what is done in *fine-grained version control,* a term that was introduced in Orwell (Thomas and Johnson 1988). It refers to version control at the function or method level rather than at the class or file level (Magnusson and Asklund 1996). This means that programmers are able to track changes between different versions of methods, as well as at the file level.

A version control and configuration management system that does allow for such a precise degree of version control is COOP/ Orm (Magnusson and Asklund 1996) shown in Figure 2.1. The upper portion of the figure labeled "versions" shows the version history of a software project. We see seven different versions of the system. The main branch consists of version 1, 2, 4, 5, and 7 while versions 3 and 6 are side branches. The arrows going from version 3 to 5 and 6 to 7 indicates a merging of different versions. In the bottom of the figure, versions 2 and 7 are being compared.



```
Procedure PutChar(ch);
   character ch;
begin
   TextString(ix):=ch;
   ix:=ix+1;
end --PutChar--;
```

This class implements the type text. A text value is a string, i.e. an ordered sequence (possible empty) of characters. The number of characters is called the "length" of ue.

Figure 2.1: Changes are presented at two different levels: on a version-by-version basis and on the level of the individual function (Magnusson and Asklund 1996).

We see a folder in the lower part of the figure showing how a part of the system, the class “Text” differs, between version 2 and version 7. There is a “!” beside the name of the class indicating that this class was modified somehow between version 2 and version 7. The different procedures for “Text” and the documentation for the class are shown within the folder for the class. We also see that documentation has been added to the class between versions, and that the procedure “SetPos” was added; both additions are indicated by the “+”, while the modified procedures “Pos” and “PutChar” are indicated by the “!”. Procedure “GetChar” is identical between the versions so that there is no symbol beside the name of the method.

Within the folder for class Text, there is another folder that provides the details of how procedure "PutChar" has changed i.e., documentation has been added to the procedure. We see that the actual code for PutChar is displayed in the folder called "Body" where additions are depicted with underlining and deletions by strikethroughs.

A second drawback of many version control and configuration management systems is that they depict only change details at a file-by-file level. With many changes, it might be useful for a person to get an overview of them to help one decide what documents and document parts that one should examine in detail. Yet there is no automated support for tracking changes to a document as a whole let alone for a collection of related documents. Doing so could be a very useful mechanism for helping a person track changes in (say) a software system that consists of many constantly changing source text files.

A third problem is that most version control systems list changes out of context, usually through a report presented separately from the changed documents. This means that one must look at a report item and try to figure how it relates to particular changes in the document itself e.g., as in the case of Diff (see Section 2.2). Context is often missing, and people must 'navigate' between the report and document. COOP/Orm is an exception. As Figure 2.1 shows (in the folder called "Body"), the system does display changes inside the document (source code) right at the point where the change was made. This way, the reader can get a better sense of the context supporting the change i.e., the operations performed by that portion of the code, and other changes made near it. Within this context different types of changes are indicated with different mechanisms such as underlining and strike-throughs.

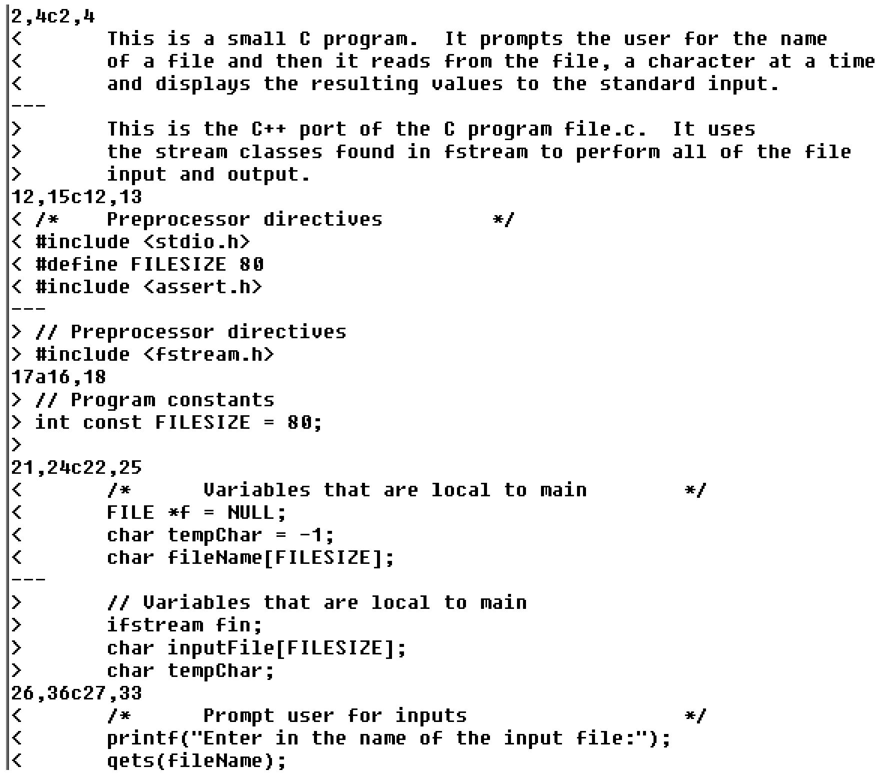
A final serious problem is that many of the systems mentioned so far are fairly old and were developed for managing ASCII-based text files. Modern computers, however, deal extensively with non-ASCII, binary files containing *rich text* (text with formatting instructions), music, photographs, and images. Although some systems such as CVS (Berliner 1990) and Visual SourceSafe (Microsoft 1992) do allow for rudimentary management of binary file versions, they work best for text files. This is because these systems cannot compute or display the actual differences between two binary files. Most use a text-based differencing algorithm that cannot make sense of binary data whose meaning depends largely upon the application and even the hardware that is used to create it. Thus they can only report that the versions differ, which provides little useful information. To display differences between binary files in a meaningful way, the version control system would need to know the structure of the binary files it handles. Given the prolific use of binary files, often with proprietary or undocumented internal structures, it is simply impossible for a version control system to handle all binary files in a robust and generic way. Consequently, people now rely on explicit documentation of changes made by the author to fully understand the differences between versions of a binary file, or they must hope a particular application knows enough about the binary file to meaningfully present information about changes.

2.2 Displaying change in text-based systems

In the first subsection below I describe the early systems for tracking and managing changes such as Diff. As mentioned in the previous section, many of these systems would represent changes separately from the changed documents, which often made change tracking difficult. In the next subsection I describe some of the later systems, such as Word (Microsoft 1983), which would imbed information about changes right in the document itself. The next subsection deals with cases when a document becomes so large that a 'bird's-eye' overview of changes becomes necessary.

2.2.1 Text based differencing

Most systems organize change information into versions and track where (e.g., the line and character position) changes have occurred. A typical example system is Diff (Hunt and McIlroy 1975), which can be used to compare and display line-by-line changes between two different versions of ASCII files. If these systems are to be useful, a person must also be able to visually compare differing versions so that they can see and understand exactly what changes were made. As will be seen in the example illustrated in Figure 2.2, this is one of the major drawbacks of Diff as a tool for supporting change awareness.



```
2,4c2,4
<       This is a small C program.  It prompts the user for the name
<       of a file and then it reads from the file, a character at a time
<       and displays the resulting values to the standard input.
---
>       This is the C++ port of the C program file.c.  It uses
>       the stream classes found in fstream to perform all of the file
>       input and output.
12,15c12,13
< /*    Preprocessor directives         */
< #include <stdio.h>
< #define FILESIZE 80
< #include <assert.h>
---
> // Preprocessor directives
> #include <fstream.h>
17a16,18
> // Program constants
> int const FILESIZE = 80;
>
21,24c22,25
<       /*      Variables that are local to main        */
<       FILE *f = NULL;
<       char tempChar = -1;
<       char fileName[FILESIZE];
---
>       // Variables that are local to main
>       ifstream fin;
>       char inputFile[FILESIZE];
>       char tempChar;
26,36c27,33
<       /*      Prompt user for inputs                  */
<       printf("Enter in the name of the input file:");
<       gets(fileName);
```

Figure 2.2: Comparing two versions of a simple file and console I/O program with Diff.

The output of Diff, seen in the figure, deserves some explanation. Changes between files are shown on a line-by-line basis in the form of a linear list. Diff outputs a left angled bracket "<" in front of lines from the first version that do not match any lines in the second version. The right-angled bracket ">" is in front of lines in the second version that do not match any lines in the first version. Lines common to both are not displayed. The problem is obvious: changes are shown out of context. Although Diff produces compact output, readers can find it hard to understand the changes because they must recall from memory what the surrounding lines of text are like in order to recover the context of the changes. Often readers must relate Diff output to the two different source files.

Furthermore, if only one character in a line differs between versions, Diff will show the entire line as being different between versions leaving the reader to manually compare the lines in order to determine exactly how they differ. Not surprisingly, if many changes have been made between versions, the reader ends up spending much time reading and re-reading Diff output to determine what has changed. In addition, Diff does not output any additional cues to help differentiate between additions, deletions, modifications or authorship, or the sequence of changes. Because of all of these limitations, one finds Diff unsatisfactory for even the simple example given in Figure 2.2 that has few changes to be compared.

With a large system that is being modified constantly by many different programmers, tracking changes on a line-by-line basis is a daunting task. Not all the changes will be relevant to a given programmer so that each person must wade through them all in order to determine which ones apply to his or her work. Eventually the issues associated with displaying change information in text-based collaborative environments began to receive research attention, and as a result later systems improved this display. These will be discussed shortly in Subsections 2.2.2 and 2.2.3.

2.2.2 Extensions to the differencing algorithms: presenting changes at the document level

A better approach to differencing is to display rich change information within the text document itself. Some time after the advent of graphical word processors that use rich text, non-programmers demanded effective ways of tracking and viewing changes between document drafts. Consequently common word processors such as Microsoft Word began to include change awareness techniques using in-line change displays, change indicators and annotations. The types of changes tracked are the addition, deletion or modification of items in the document. Figure 2.3 shows a Word document with the change tracking function turned on. We see that modifications are treated as an insertion immediately followed by a deletion. We also see a vertical bar in the left margin beside lines that have changed, which helps people spot lines with small changes.

Text that is the same between versions is rendered in the original black; text that differs between versions is rendered in different colors, one for each editor. The example in the figure highlights changes by coloring the changed text red. Text added to the version is rendered with an underlined typeface, while text that has been deleted is rendered with a strike-through typeface. Changed text that has been annotated with comments by an author is highlighted in yellow, and mousing-over it will reveal the annotation in a small, transient, pop-up window.

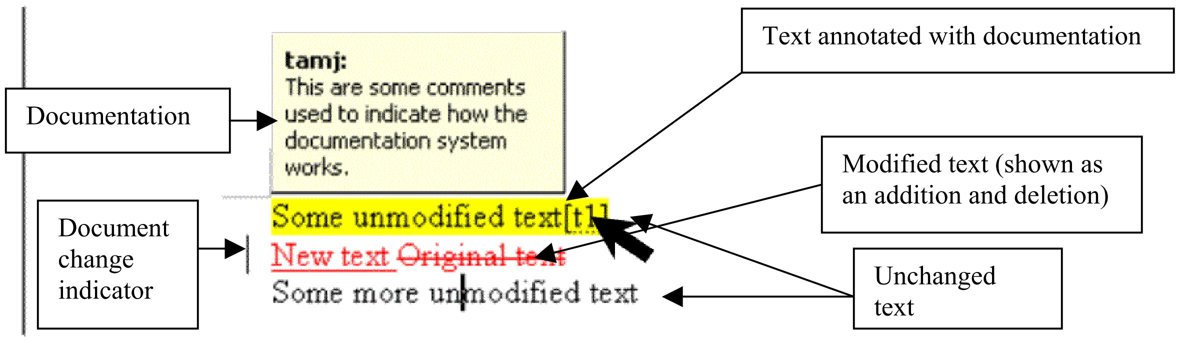
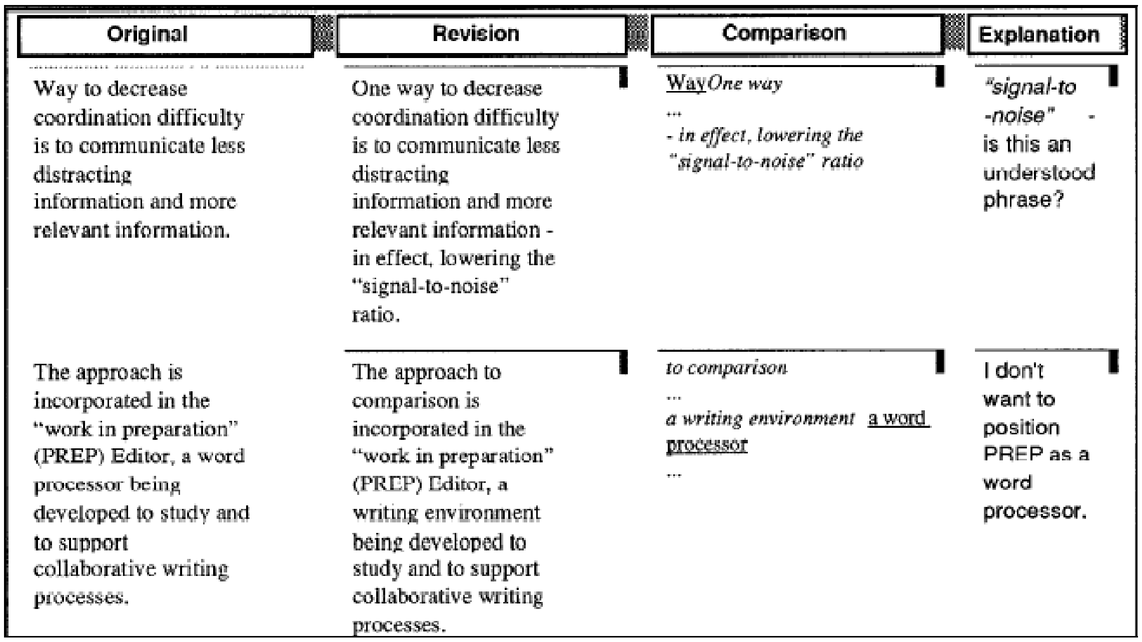

Text annotated with documentation
Documentation
tamj:
This are some comments used to indicate how the documentation system works.
Modified text (shown as an addition and deletion)
Document change indicator
Some unmodified text[t1]
New text ~~Original text~~
Some more unmodified text
Unchanged text

Figure 2.3: In Microsoft Word (Microsoft 1983), the modification of a string of text is treated as an insertion followed immediately by a deletion of the old text.

Unlike version control systems, which compare saved snapshots, Word allows the reader to track changes in real time. As a change is made to the document, a corresponding change indicator will be immediately displayed to the user. In the older version control systems, one had to leave the text editor and run a separate program to track changes.

A more sophisticated system that displays changes inline is Flexible Diff (Neuwirth, Chandhok, Kaufer, Erion, Morris and Miller 1992), illustrated in Figure 2.4. Flexible Diff is an extension of the PREP editor (Cavalier, Chandhok, Morris, Kaufer and Neuwirth 1991; Neuwirth and Kaufer 1989; Neuwirth, Kaufer, Chandhok and Morris 1990). In this figure, we see how four columns are used to communicate the nature of a change. The original and modified text are in the first and second columns respectively. Column 3 shows only the differences between these versions while Column 4 shows annotations added by the author that explain the changes.

This last column of annotations was included to alleviate the frustrations that authors would sometimes experience when they encountered unexpected changes made by other co-authors (Cross 1990). Annotations also help draw a reader's attention to a particular change (Neuwirth, Chandhok, Kaufer, Erion, Morris and Miller 1992). If we go back and compare the output from the original form of Diff in Figure 2.2 with the output of Flexible Diff in Figure 2.4, it is obvious that this four-column format makes it much easier for the reader to find, track and understand changes. The cost is that the original document format (e.g., its formatting, use of white space, etc.) is lost in this view.



Original	Revision	Comparison	Explanation
Way to decrease coordination difficulty is to communicate less distracting information and more relevant information.	One way to decrease coordination difficulty is to communicate less distracting information and more relevant information - in effect, lowering the "signal-to-noise" ratio.	Way *One way* … *- in effect, lowering the "signal-to-noise" ratio*	*"signal-to -noise"* - is this an understood phrase?
The approach is incorporated in the "work in preparation" (PREP) Editor, a word processor being developed to study and to support collaborative writing processes.	The approach to comparison is incorporated in the "work in preparation" (PREP) Editor, a writing environment being developed to study and to support collaborative writing processes.	*to comparison* … *a writing environment* a word processor …	I don't want to position PREP as a word processor.

Figure 2.4: The four-column view of changes provided by Flexible Diff (Neuwirth, Chandhok, Kaufer, Erion, Morris and Miller 1992).

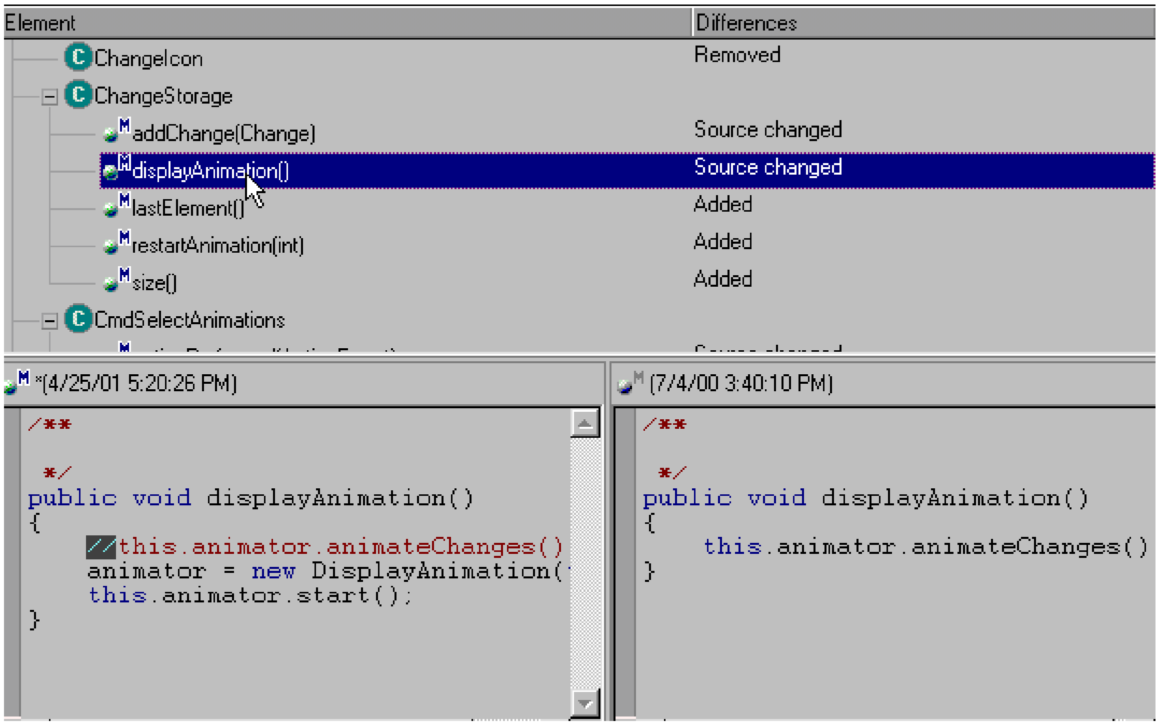
The "flexible" part of Flexible Diff' refers to the fact that this algorithm, unlike its predecessors, can vary the granularity in the display of changes. A reader can specify the "grain size" (Neuwirth, Chandhok, Kaufer, Erion, Morris and Miller 1992) as the length of a text string that must be changed before the change is reported. Within this grain size, the reader can also specify that changes should be shown only where some proportion of the grain was changed. For instance, if the reader chooses to display changes at the 100% level of the word grain size, then he or she will see change information displayed for words that are not exactly the same between versions. Setting the percentage of grain size changed below 100% means that some changes may not be displayed to the reader. The reader can also choose to ignore changes made to white space (spaces, tabs, new lines). Combined, these features give the reader an opportunity to filter out the display of trivial changes about which he or she need not be concerned.

This filtering of changes is important. Nachbar (1988) argues that it is not always appropriate to report all changes for some types of tasks. Most models of reading assume that reading requires that the reader must expend some of his or her attention in order to derive meaning from text. Since a reader's attention is limited, it must be allocated among tasks (Samuels and Kamil 1984). When multiple tasks demand more attention that one can afford, the tasks cannot be performed in parallel and instead one must allocate his or her attention carefully with some form of "attention switching" (Neuwirth, Chandhok, Kaufer, Erion, Morris and Miller 1992). However, this attention switching puts a heavy load on short-term memory and may interfere with recall (Kahneman 1973).

Simply displaying change information invites the reader to pay attention to it: this could *distract* the reader, and steal this person's attention away from his or her primary task e.g., when he or she is more concerned with reading the document rather than tracking changes between different versions. Lesser and Erman (1980) define distraction in this context as the degree to which a person's focus can be shifted. Neuwirth, Chandhok, Kaufer, Erion, Morris and Miller (1992) further specify that "positive distraction shifts the agent to tasks that are more useful; negative distraction shifts the agent to tasks that are redundant or diversionary" (pp. 149). Thus, they advise that the only changes that should be reported are ones that "…will reduce negative distraction and increase positive distraction" (pp. 149). Unfortunately, whether the display of a given change will increase positive distraction depends largely on the reader's goals for reading the document. It is for precisely this reason that Flexible Diff that permits the reader to adjust the degree to which information about changes are displayed.

VisualAge (IBM 1991) is an IDE (Integrated Development Environment) produced by IBM for writing software in many different programming languages such as Java (Sun Microsystems Inc. 1996), C++, Smalltalk etc. Its integrated version control and differencing function also overcomes some of the shortcomings of Diff. In the top window of Figure 2.5, two different versions of class are compared, and unlike Diff, an indicator is provided next to classes, methods and data to indicate the type of change that occurred between versions: removals, additions and modifications are tagged by 'Removed', 'Added' and 'Source changed' respectively.

VisualAge also allows changes to be viewed and filtered at various levels of the hierarchy: at the package, class or method level. In the case of viewing changes, multiple lower-level changes are combined and described as a modification. For example, if a person were viewing changes of a package that had some classes added and deleted, VisualAge would describe the change made to the package as a modification. Users can then select particular items for detailed viewing.



Element	Differences
ChangeIcon	Removed
ChangeStorage	
addChange(Change)	Source changed
displayAnimation()	Source changed
lastElement()	Added
restartAnimation(int)	Added
size()	Added
CmdSelectAnimations	

*(4/25/01 5:20:26 PM)

```
/**

 */
public void displayAnimation()
{
    //this.animator.animateChanges()
    animator = new DisplayAnimation(
    this.animator.start();
}
```

(7/4/00 3:40:10 PM)

```
/**

 */
public void displayAnimation()
{
    this.animator.animateChanges()
}
```

Figure 2.5: Version differencing of Visual Age (IBM 1991).

For modifications, Visual Age highlights the bottom window for each individual change, one at a time. In the top of Figure 2.5, the user has selected the method 'displayAnimation', which has had its source changed. The bottom left part of the figure shows the source code for the newer version of this method while the bottom right part of the figure shows the source code for the older version. This change information is presented in an in-line fashion as opposed to the separate fashion used by Diff. For example, the figure shows that there are three differences in the method "displayAnimation", where some code was 'commented out' (by preceding it with a pair of backslashes), a variable was initialized, and a function call was added. The figure shows how the first part of a line is highlighted (the slashes) indicating that this is the part of the line that is somehow different between versions. Notice, however that Visual Age does not explicitly indicate that this line of documentation was added between versions (in the bottom part of window). Viewers must compare the two versions themselves in order to make this determination.

In summary, all these systems show how differences are better displayed. We saw how some (but not all) systems show changes within the text, allow side-by-side comparison of changed items by marking changes, include author annotations, and allow a reader to selectively adjust the level of how changes are filtered and displayed.

2.2.3 Presenting an overview of changes

Many of the systems discussed so far have focused on displaying the individual changes within a document. The specific changed parts of a document (paragraphs, sentences, words or characters) are marked with some form of additional information to communicate the change. What is missing is a way to communicate an overview about changes made to the entire document, which is necessary if one wants to get a general sense of what has changed and where one should start looking for details in a large document or in a collection of related documents.

One approach shows readers a graphical overview of what has changed. A good example is Hill and Hollan's (1992) 'read-wear' and 'edit-wear' overview that uses the metaphor of how objects in the real world tend to wear out from use. Figure 2.6 illustrates their "attribute-mapped scroll bars" (Wroblewski, Hill and Mccandless 1990), where indications of wear are displayed as marks within a scroll bar. The marks are mapped onto a document in a position relative to the line that was changed. The more that a region of a document has been read or edited, the longer the mark will be. Since the length of the scroll bar matches the length of the document and relative positions in the scroll bar correspond to a relative position in the document, the scroll bar itself provides a rough birds eye overview of the quantity of changes and where changes within a document are located. As well, multiple categories of wear indicators can be shown in different columns of the scroll bar to communicate different information, as illustrated in Figure 2.7. Here, we see 3 columns, each displaying the read history of an individual author.

Although attribute-mapped scroll bars are useful for showing a compact overview of where changes in a document lie, they do not communicate how the document changed (e.g., what words differ) or why (i.e., they do not give authors and editors the chance to annotate the document with the reasons behind changes). One would thus expect this to be integrated within another system that does provide this information.

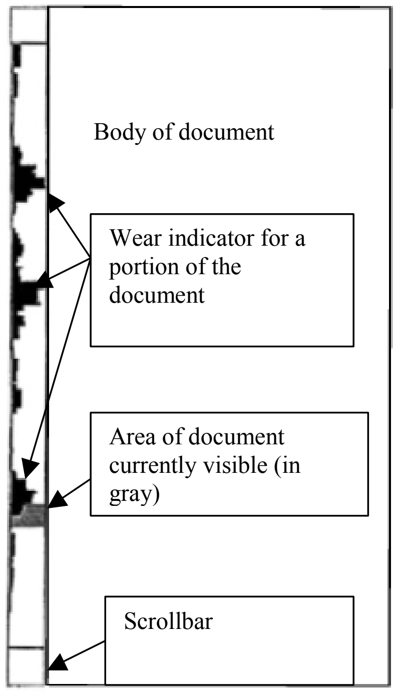

Body of document
Wear indicator for a portion of the document
Area of document currently visible (in gray)
Scrollbar

Figure 2.6: Read-wear/Edit-wear showing one category of wear. Adapted from Hill and Hollan (1992).

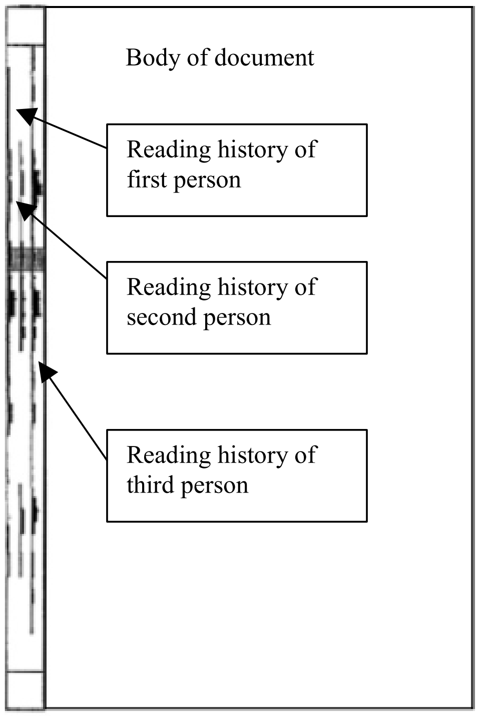
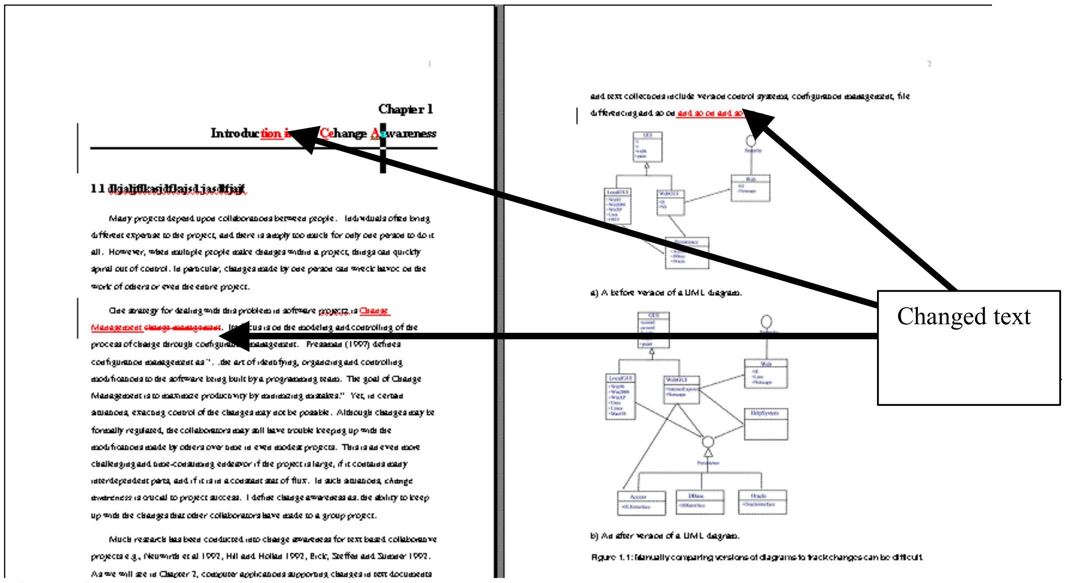

Body of document
Reading history of first person
Reading history of second person
Reading history of third person

Figure 2.7: Using multiple wear indicators to show which parts of the document have been read by different people. Adapted from Hill and Hollan (1992).

A different approach to gaining an overview uses zooming. We previously saw how Microsoft Word provided a detailed in-line display of changes. Word also provides the reader with the ability to see an overview of a long document simply by letting a person ‘zoom-out’. While no additional change information is supplied, the differing colors used for fonts and graphics provide a sense of what has altered. For example, in Figure 2.8a, we see two pages of Chapter 1 of this thesis, where pages are shown at 50% of its original size. Figure 2.8b shows another chapter at 10% of the original document size. Although many more pages can be represented with a smaller document size the color indicators used to represent changes are smaller and thus are more likely to be missed.



Chapter 1

Introduc~~tion~~ in Ce hange Aa wareness

1.1 dkjaljflkasjdflajsd.jasdlfjaj

Many projects depend upon collaborations between people. Individuals often bring different expertise to the project, and there is simply too much for only one person to do it all. However, when multiple people make changes within a project, things can quickly spiral out of control. In particular, changes made by one person can wreck havoc on the work of others or even the entire project.

One strategy for dealing with this problem in software projects is Change Management ~~change management~~. Its focus is on the modeling and controlling of the process of change through configuration management. Freeman (1997) defines configuration management as ". . .the art of identifying, organizing and controlling modifications to the software being built by a programming team. The goal of Change Management is to maximize productivity by minimizing mistakes." Yet, in certain situations, exacting control of the changes may not be possible. Although changes may be formally regulated, the collaborators may still have trouble keeping up with the modifications made by others over time in even modest projects. This is an even more challenging and time-consuming endeavor if the project is large, if it contains many interdependent parts, and if it is in a constant state of flux. In such situations, change awareness is crucial to project success. I define change awareness as the ability to keep up with the changes that other collaborators have made to a group project.

Much research has been conducted into change awareness for text based collaborative projects e.g., Neuwirth et al 1992, Hill and Hollan 1992, Eick, Steffen and Sumner 1992. As we will see in Chapter 2, computer applications supporting changes in text documents

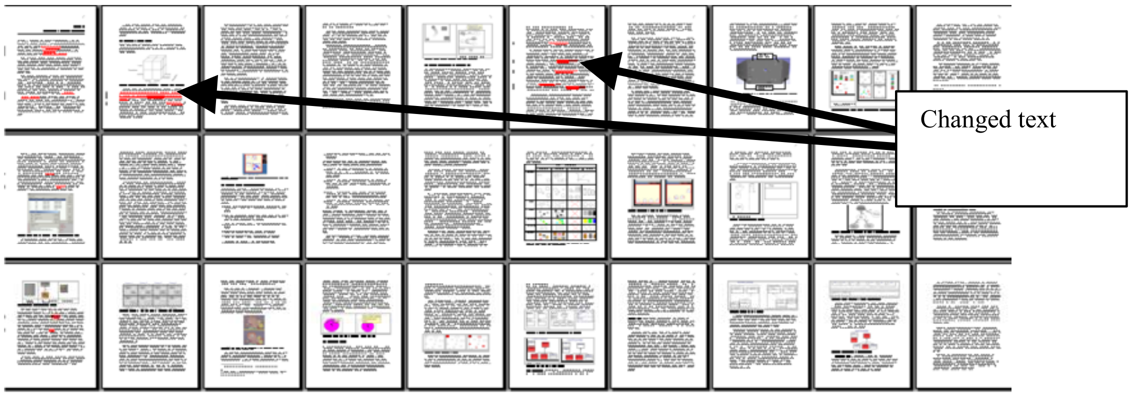
and text collections include version control systems, configuration management, file differencing and so on and so on and so on.

a) A before version of a UML diagram.

b) An after version of a UML diagram.

Figure 1.1: Manually comparing versions of diagrams to track changes can be difficult.

a) 50% of original size



b) 10% of original size

Figure 2.8: Overview of a portion of different chapters of this thesis displayed with Microsoft Word (Microsoft 1983) at 50% (a) and 10% (b) of the original size.

The Seesoft system better leverages a zoomed out overview of changes made over a collection of related documents (Eick, Steffen and Sumner 1992). Illustrated in Figure 2.9. Seesoft provides an overall view of a software system in a "reduced representation" of the code. Each column of the overview represents a single file; each line in a column represents a line in the file. The length of each line in the column shows the indentation and line length. The height of each column shows the size of the file and files that are too large to represent in a single column are wrapped to multiple columns. This overview representation of the project is similar to the attribute mapped scroll bars (Wroblewski, Hill and Mccandless1990) except we can now see several documents at a time (up to a maximum of 50,000 lines of code or so).

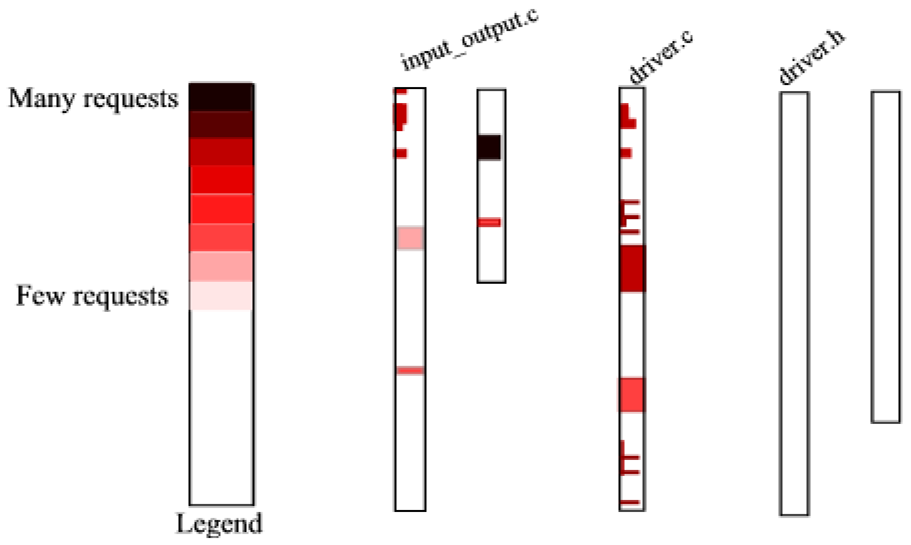


Figure 2.9: A re-creation of Seesoft (Eick, Steffen and Sumner 1992). It shows the modification requests for bug fixes made to a software project.

Unlike attribute-mapped scrollbars, users can see a detailed view of particular lines of code on demand. Whenever a user wanted to read the code that corresponded to the parts of the overview display, he or she could do so by clicking on a column. This would pull up a magnification window, which would show the actual code.

Seesoft adds value to the bird’s eye view by using line coloring to indicate characteristics of the code represented, such as its age, the developer responsible for it, or the number of times it has been tested. For instance, in Figure 2.9, the brightness of the color red (as chosen through the legend on the left) is mapped to the number of modification requests made to the code. Thus we see those portions of the project shaded dark have received more modification requests than those that are more lightly shaded.

Using color, Seesoft only communicates one characteristic of the code at a time. However, the system does allow the user to change quickly the mapping between color and the different types of statistics through *Direct Manipulation* techniques

(Shneiderman 1983) i.e., the legend on the left can be manipulated to represent other attributes.

The systems seen so far portray fairly literal representations of change. The Theme River prototype (Havre, Hetzler and Nowell 2000) differs as it abstracts the minute details of changes. This system aggregates all changes made to a collection of documents and displays them as trends that are taking place in the collection over time (Figure 2.10). Theme River is based upon the principles of the Gestalt school of Psychology and the belief that “the whole is greater than the sum of the parts” (Koffka 1935). It capitalizes on the notion that people can use their perceptual skills to see high-level relationships between multiple related documents. Its visualization uses a metaphor of a river that travels horizontally across the screen (Figure 2.10). Time varies along the x-axis, while the strength of a trend (e.g., the number of times that a particular word appeared in a collection of documents) is shown along the y-axis. This visualization provides quick and aggregated representations of the rise and fall of trends (changes over time).

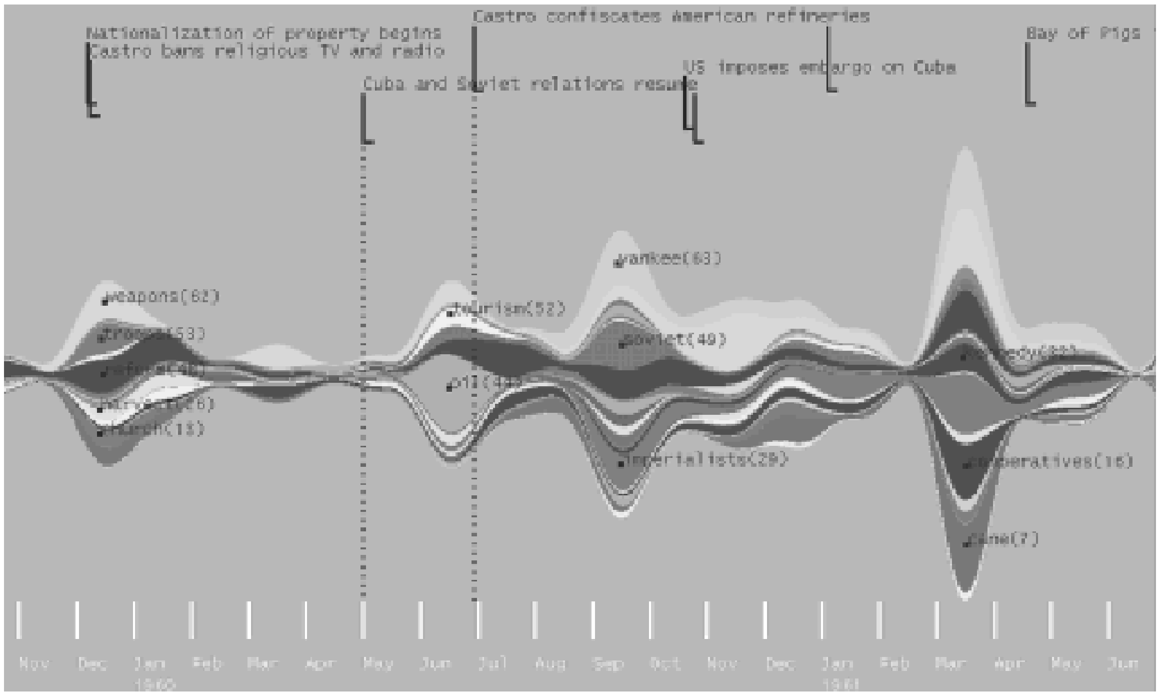


Figure 2.10: The Theme River (Havre, Hetzler and Nowell 2000) system represents changes as rivers that flow over time.

Usability tests found that while users liked abstracting from individual documents to the collection of documents as a whole, they also wanted to see individual documents on demand (Havre, Hetzler and Nowell 2000). Consequently while Theme River and perhaps other overview visualizations are useful for communicating high-level change information, other levels of detail are required. This follows Shneiderman (1996), who suggests in his InfoVis mantra, overview first, details on demand.

2.2.4 Existing portrayals of changes in text-based systems

Three key points must be kept in mind with regard to the systems I have discussed thus far. First, the representation mechanism used to display and thus communicate rich change information is an important part of the system and has a tremendous impact on the usability of the system. Second, because modern text editing systems no longer work exclusively with text, a change tracking system must also deal with non-text elements (e.g., formatting, pictures) in documents. Third, changes should be displayed on multiple levels: from the *macroscopic* perspective of the entire collection of documents or individual document right down to a *microscopic* view of changes shown on an item-by-item basis. Ideally, the granularity of changes shown should be easily and rapidly adjustable in order to suit different change-related tasks.

2.3 Change awareness in existing two dimensional graphical systems

All of the systems discussed so far deal primarily with sequential text. There has been appallingly little research into the problems associated with tracking changes made to two-dimensional drawings. This is even more astounding in light of the fact that graphics are used in a wide variety of applications. For example, graphical editing packages such as paint and structured drawing packages are used pervasively for a wide variety of familiar tasks such as constructing technical drawings, organizational charts, network diagrams, flow charts or architectural diagrams (Kurlander 1993). Less familiar are concept maps, a semantically rich method of visual communication that has "…a history of use in many disciplines as a formal or semi-formal diagrammatic technique"

(Gaines and Shaw 1995: pp. 323). Similarly, UML diagrams are the standard design and modeling language for software (Booch, Rumbaugh and Jacobson 1999; Fowler and Scott 1997; Rumbaugh, Jacobson and Booch 1999). In groupware systems such as TeamWave (TeamWave Software Ltd. 1997) a room-based metaphor is used to spatially relate and reveal the rich, natural social interactions of the everyday world within collaborative virtual work (Greenberg and Roseman 2001).

One example system that provides only crude change awareness support for graphics is Microsoft Word. Despite the rich support for text, Word has poor change support for images. Usually, the entire image is treated as though it were a single character, and when any change—large or small—is made to the picture, the entire picture is rendered with a change indicator similar to those applied to text (Figure 2.11). As with the case of changes to text, there is a vertical line added to the changed image to better indicate which part of the document was changed. Because the particular example shown is a vector graphic, Word also shows a small gray "ghosted" shadow to indicate where the two images differ: (the right peg leg of the cartoon character). With other types of images, the reader is left to his or her own devices to determine how the graphic has changed.

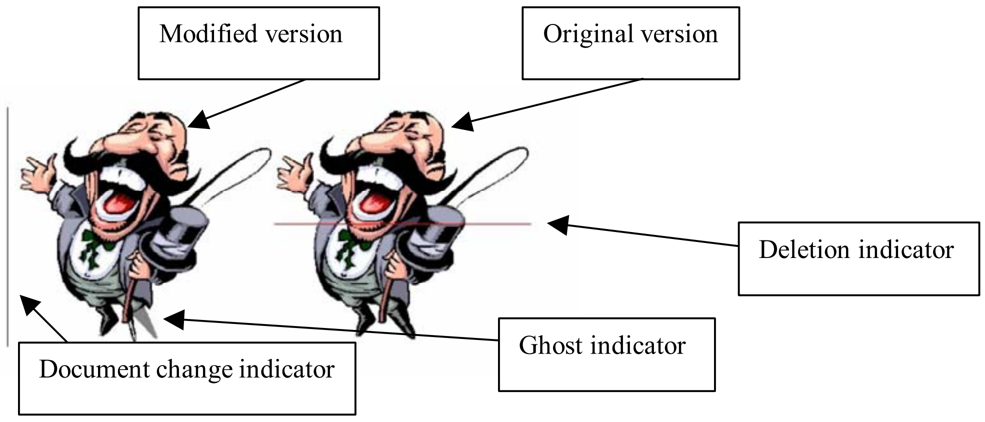
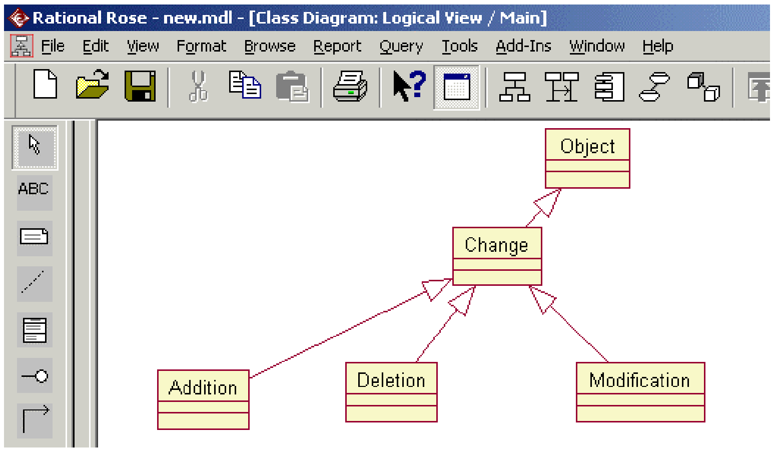


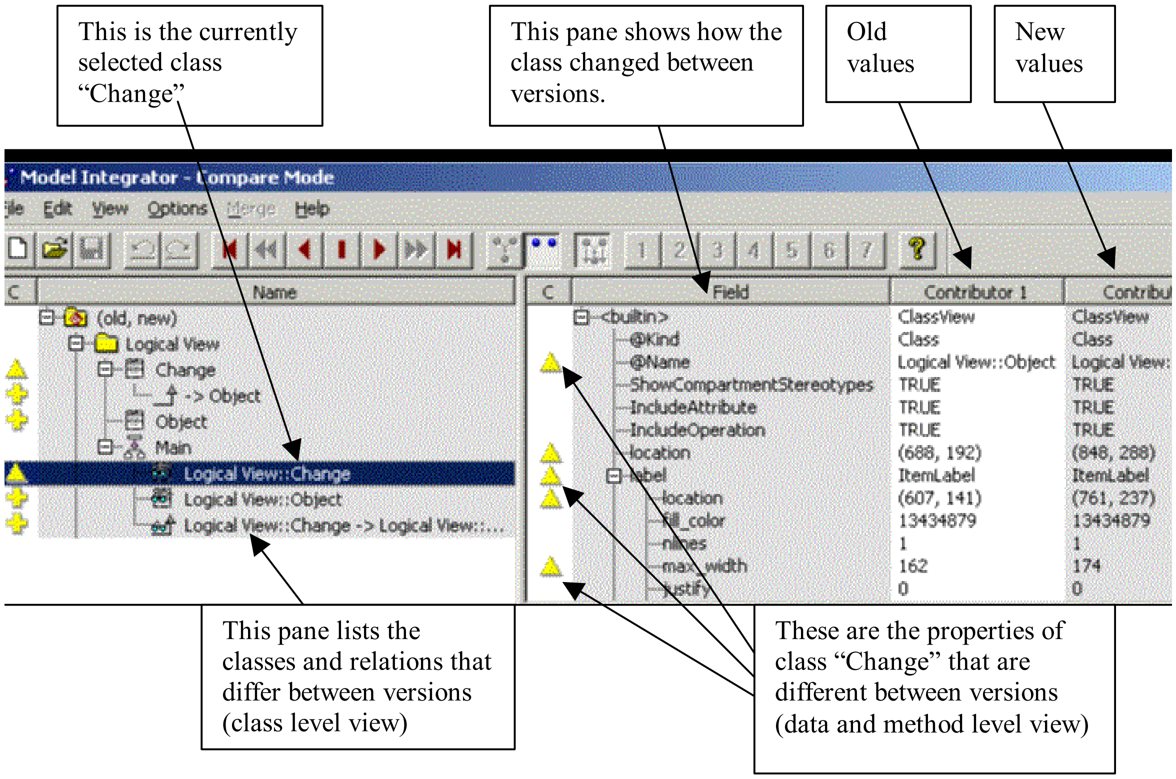
Figure 2.11: The modification of a picture illustrated by showing the modified version (left) and the original version (right with a red strike-through) in Word (Microsoft 1983).

An example of a popular, specialized 2D graphical application is Rational Rose (Rational Software Corporation 1991), a UML editor used in the specification and design of software systems (Figure 2.12a). The typical working view of classes in Rational Rose is that of a UML diagram where classes can be created and edited. Although there are several aspects to these classes that aren't represented using this graphical view, the advantage to having a diagrammatic representation is that it is easy to spot the relations between classes. Changes made to this diagram, however, are not shown in this view: instead a mostly text oriented, Model Integrator view provides change information in a separate window (Figure 2.12b). Its purpose is to help software engineers determine the differences between two branch versions of the project and to merge these branches rather than for change tracking.

Figure 2.12 illustrates a typical scenario of its use. A developer returns to the classes in Figure 2.12a some time after having last worked on it. In the interim, others have made changes to this model, and now the developer wishes to determine which classes have changed and how they have changed. Unfortunately, the working view presented in Figure 2.12a does not give any information regarding these changes, and so instead, the designer must access the Model Integrator in Figure 2.12b, which looks entirely different from the working view. It has a pane on the left hand side that displays the classes in the model as a tree, which differs considerably from the working view's graphical depiction. Although it can be argued that UML diagrams contain information that isn't visually represented (nor is there a need for it to be represented visually) some information is represented *best* in a visual form (Larkin and Simon 1987). Moreover, by presenting the information about changes made to the UML diagrams in a non-diagrammatic form, some viewers may miss the context in which the changes were made.


Rational Rose - new.mdl - [Class Diagram: Logical View / Main]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
ABC
Object
Change
Addition
Deletion
Modification

(a) An example of a class diagram in the working view provided by Rational Rose (Rational Software Corporation 1991).


This is the currently selected class "Change"
This pane shows how the class changed between versions.
Old values
New values
Model Integrator - Compare Mode
File Edit View Options Merge Help
1 2 3 4 5 6 7
This pane lists the classes and relations that differ between versions (class level view)
These are the properties of class "Change" that are different between versions (data and method level view)

C	Name
	(old, new)
	Logical View
	Change
	-> Object
	Object
	Main
	Logical View::Change
	Logical View::Object
	Logical View::Change -> Logical View::...

C	Field	Contributor 1	Contribu
	<builtin>	ClassView	ClassView
	@Kind	Class	Class
	@Name	Logical View::Object	Logical View:
	ShowCompartmentStereotypes	TRUE	TRUE
	IncludeAttribute	TRUE	TRUE
	IncludeOperation	TRUE	TRUE
	location	(688, 192)	(848, 288)
	label	ItemLabel	ItemLabel
	location	(607, 141)	(761, 237)
	fill_color	13434879	13434879
	nlines	1	1
	max_width	162	174
	justify	0	0

(b) The Model Integrator view provided by Rational Rose (Rational Software Corporation 1991).

Figure 2.12: The regular working view of UML diagrams (a) and the Model Integrator view (b) which is the only (indirect) support provided by Rational Rose (Rational Software Corporation 1991) to track changes.

Another possibility of viewing graphical changes is through a graphical history. At its simplest, one could perhaps track changes in conventional drawing systems by using the undo/redo feature to trace backwards and forwards through the history of modifications that have been made to a picture over time. This is hardly a practical solution if many different people have made many changes and besides, few conventional systems maintain undo/redo information between sessions.

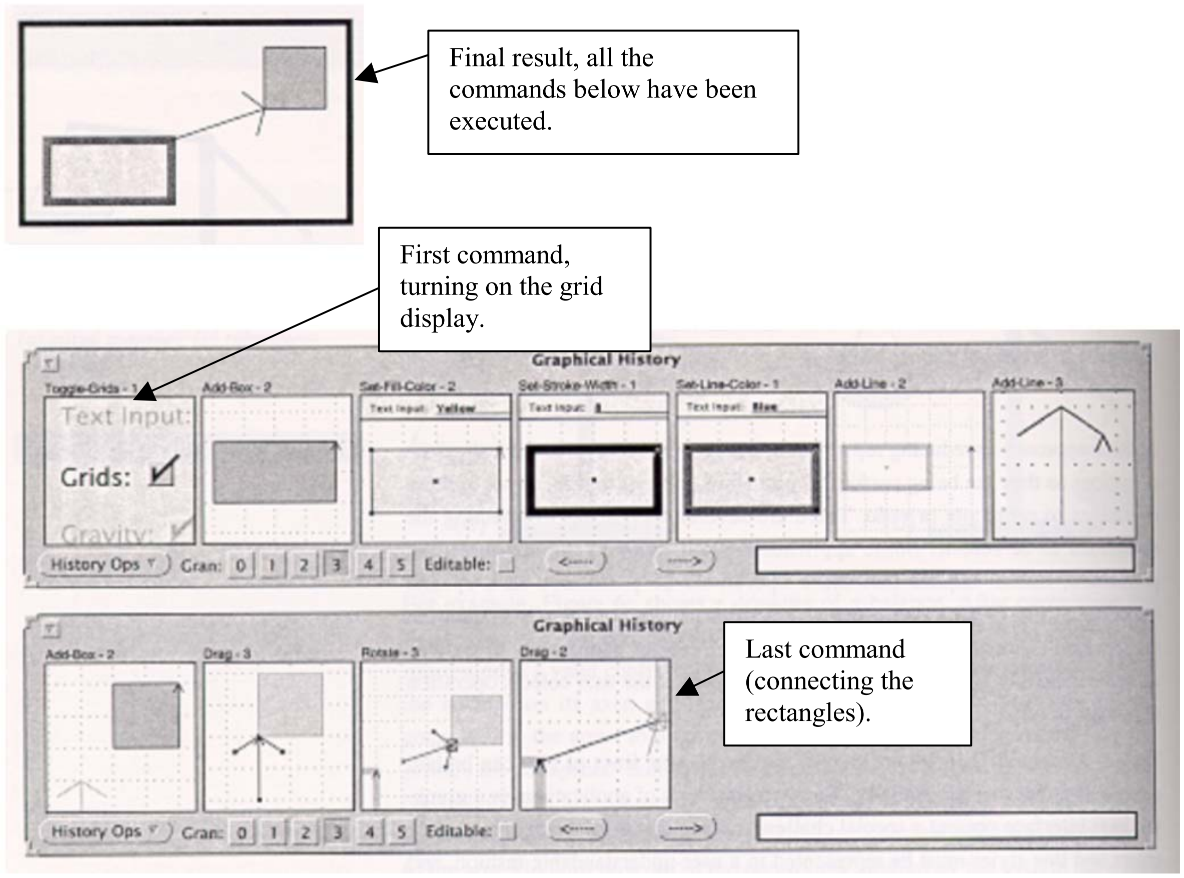


Figure 2.13: Editable graphical histories, showing the series of commands (bottom) needed to produce the final picture (top). Adapted from Kurlander and Feiner (1993).

A better example of how change can be supported via a graphical history is hinted at by Kurlander and Feiner's (1991; 1993) Chimera system. Chimera gives users the ability to edit a graphical history, and Figure 2.13 shows an example. The top picture shows the final completed picture as drawn by the user. The bottom picture consists of a series of storyboard panels that show the sequential progression of that person's drawing actions and their visual results. The system does not provide a panel for every action; rather, it determines when a change is significant enough to warrant its display. While this system was not developed for change awareness, the "storyboarding" technique it uses can potentially communicate change information in 2D graphical systems. I will elaborate further on how this technique can be used to represent changes in Chapter 4.

2.4 Summary

In this chapter, I briefly surveyed approaches to the management and display of change information. I started by describing the early version control systems such RCS (Tichy 1991), where changes between versions were displayed at a rather coarse granularity. I then explained how Configuration Management systems are better at tracking and managing changes in projects that involved multiple developers working on multiple files. After this I described how many of these early systems display changes in the form of a linear 'differencing' list, usually as a hard to decipher report presented separately from the changed files. Also, these systems can only handle ASCII text files, and could (at best) indicate if versions of binary files differed.

As text editors became increasingly sophisticated, better change tracking systems were developed to help authors understand the edits made to shared documents. Some systems embedded changes inline within the document as markups e.g., visual indicators like lines, highlighting, typeface changes and color differences. Other systems provide an overview of changes to indicate which parts of the document had changed, but do not show how those parts changed. Some systems do both (e.g. Seesoft), providing an overview that can be used to quickly pinpoint where changes were made while giving users the opportunity to dive into detail to see what has actually changed.

Considering how widespread two-dimensional graphical applications are, there is surprisingly little change awareness support for these types of systems. For example, there is no explicit support of change awareness built into the widely used UML editor

Rational Rose (Rational Software Corporation 1991). While the Model Integrator function can be pressed into service, as I indicated earlier, this function was meant for an entirely different task. Though not originally intended to provide change awareness information, the storyboarding approach employed in the Chimera system (Kurlander and Feiner 1991; 1993) for editing graphical histories is promising.

In the remainder of this thesis, I will draw upon some of the lessons learned from previous efforts to support change in text-based systems (such as the value of information filtering) and apply them to 2D graphical systems.   In addition, I will return to some of the more promising mechanisms, such as overviews and storyboarding.