TR 00-042

Enhancing Visibility of Annotations of Software Artifacts

Michael V. Stein

July 14, 2000

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral thesis by

Michael Vincent Stein

And have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

John T. Riedl

Name of Faculty Adviser

Signature of Faculty Adviser

Date

GRADUATE SCHOOL

Enhancing Visibility of Annotations of Software Artifacts

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Michael Vincent Stein

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

John T. Riedl, Adviser
June, 2000

ACKNOWLEDGEMENTS

I wish to acknowledge those people who helped me survive the arduous process of getting a PhD. I apologize to those whom I have inadvertently failed to acknowledge.

First, my advisor John Riedl. When contemplating a return to graduate school, I interviewed all the professors I could find, looking for an enthusiastic soul who also understood that a grad student might want to "have a life". John has turned out to be an excellent, patient, understanding, creative, and articulate advisor.

My wife, Lynn Borden, patiently endured the endless time it takes to get a PhD, especially when juggling this goal with family responsibilities. Without her support, I would have doubtless failed.

Mats Heimdahl was an excellent professor with whom to work on software engineering issues. He brought a fresh perspective to the work of this thesis, and complemented my advisor and me in many respects. Special thanks to Mats for the alternative formulation of the mathematical model of annotation in Z.

Soeren Harner, formerly of ICEM Systems, is one of the most creative practicing engineers I have met. Throughout this project, he would come up with voluminous creative ideas that helped shape the work we did.

We hired Quang Dang to work on the programming of AnnoSpec, and ended up getting to know a marvelous, hard-working professional. Libby Shoop, Jon Herlocker, and Ed Chi were all helpful in answering scores of dumb questions I had about database systems, user interfaces, and just plain how to get things done.

Years ago, I chose my first job out of college, at what is now Lucent Technologies, because it was the only offer I got that did not involve at least a little computer programming, and I was afraid of computers! So it amazes me that I am getting a PhD in Computer Scuence. I owe my overcoming of that "computer anxiety" to two of my colleagues from those days, Joanne Yasko and Lynn Pautler.

ABSTRACT

People who work collaboratively often review each other's work. They may do so by annotating work artifacts, such as documents. People seeking to understand annotated artifacts confront two problems. *Clutter* occurs when people see an overwhelming number of annotations. Clutter is a well-studied problem, whose solution involves *narrow-filtering* annotations: showing a viewer only a subset of annotations on the viewed artifact. Another, less widely studied problem is *delocalization*. Delocalization occurs when people must understand multiple parts of an artifact, and annotations on those parts, to understand the part they are viewing. For instance, a person viewing a system design document might need to understand the system requirements to understand the design. We propose that delocalization of annotations can be addressed by *broad-filtering* annotations: showing viewers of an artifact some annotations that were made to other parts of the artifact, in the context of that portion of the artifact they are viewing.

We are particularly interested in annotation of software artifacts, which is a common task in software development. We introduce a three-dimensional taxonomy for delocalization within software systems: *lateral* delocalization (between items of the artifact within the same development phase), *longitudinal* delocalization (between items in different development phases), and *historical* delocalization (between successive versions of the same item). We discuss methods for ddressing clutter and delocalization concurremtly. We develop a graph-theoretic model of annotated artifacts incorporating clutter and delocalization, prove the mathematical consistency of this model, and apply this model to various existing artifacts and systems to demonstrate its breadth of applicability.

Annotation is a vital part of software inspection, which is a well-known means of software quality improvement. We discuss how to address delocalization within software inspection. We describe the development and use of *AnnoSpec*, a software inspection tool supporting clutter reduction and visibility of delocalized annotations in accordance with our model of annotation. We conducted a pilot study in which software professionals performed inspections with AnnoSpec. Results of this study suggest that addressing delocalization has some benefits in the helping people perform software inspection effectively, providing a "proof-of-concept" for the use of delocalized annotations in software inspection.

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Background on Annotation

People have annotated written documents for centuries, even before the invention of the printing press (Cavalier et. al., 1990). In these ancient texts, notably glossed bibles (Figure 1.1), we can see certain characteristics of annotation. For instance, each annotation has a *scope* consisting of the underlying text being annotated, or of a previous annotation on that text to which it applies, and each annotation is identified by author. The positioning, size, and location of the text was used to indicate the material being annotated.

Annotation of hard-copy documents continues to this day. We annotate by hand such hard-copy documents as reports and papers written by students in courses we are teaching, and draft versions of conference papers that we write collaboratively with colleagues. Annotation remains a significant component of team collaboration as we evolve toward electronically-stored artifacts (Jones, 1995). Tools exist in general-purpose word-processing programs such as MS Word (MS Word) to permit electronic annotations. Some tools even permit annotation of generalized information on the Internet (Yee, 1998; Third Voice Corp.). In particular, we consider annotation of textual or graphical *software artifacts* - human-readable information about software systems - to be a common and important software engineering activity.

Annotators commonly annotate many types of software artifacts, such as system specifications, designs, source code, test plans, and customer documentation. They annotate artifacts in various roles such as customers, software development managers, maintenance personnel, and system architects. They annotate artifacts for many purposes, including asking for clarification or suggesting a possible defect. In this thesis, we limit our discussion to annotations meant primarily to be read by people. It is possible that in some circumstances an annotator would create an annotation to be parsed by a program, rather than read by a person. Such annotations are beyond the scope of this thesis.

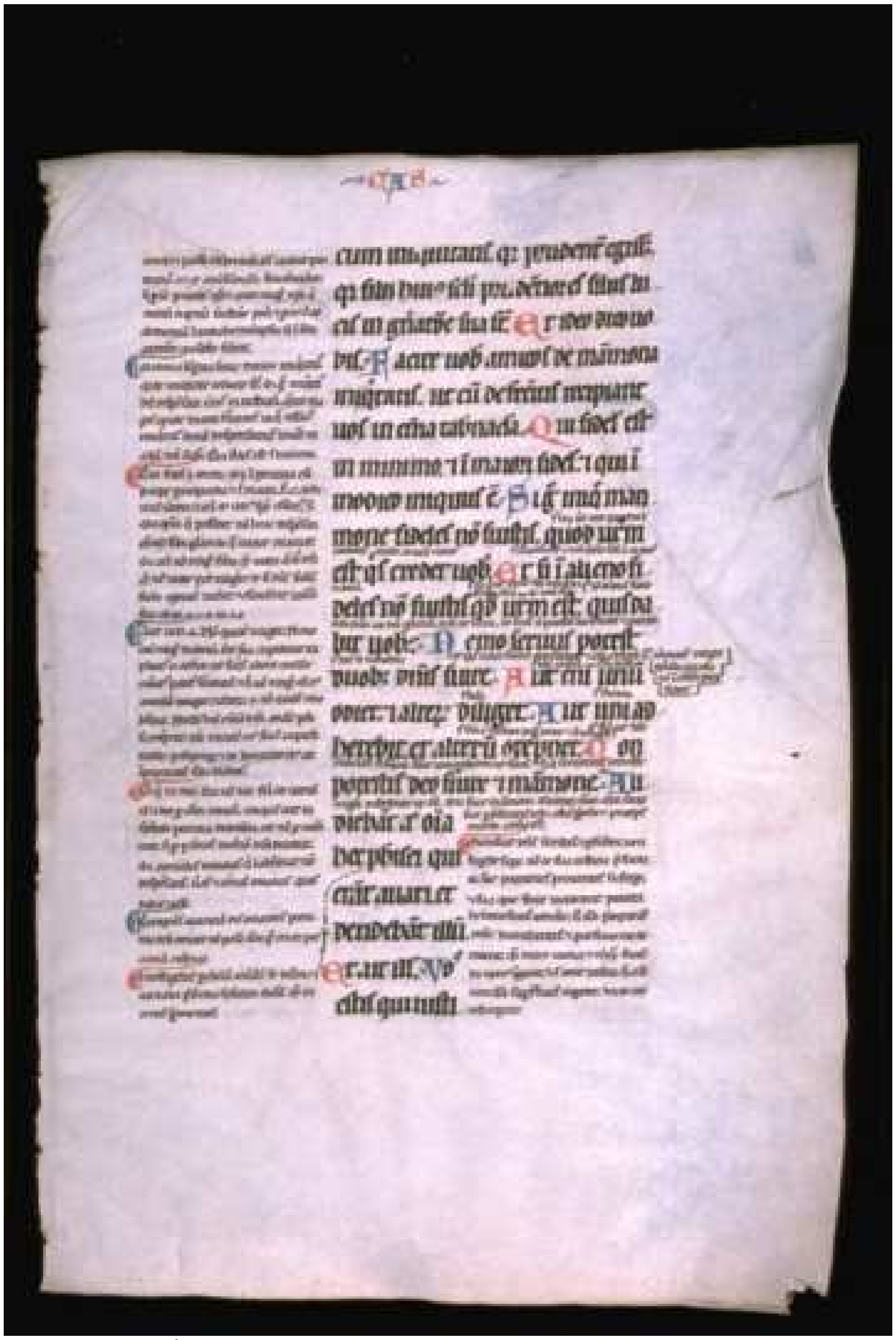


Figure 1.1: 12th-century French glossed bible. (Columbia University LibraryWeb) The large type is the source document. Space was left both for general annotations of broad scope, as in the left column and lower right; and for small, interlinear annotations, visible between the lines and in the right margin (Cavalier et. al., 1990).

Automated annotation tools are used in diverse areas such as annotating source code to explain design rationale for software maintenance (Lougher and Rodden, 1993a), and performing formal software inspection of a wide range of systems (MacDonald and Miller, 1999). The default behavior in many tools is to assume that viewers of an artifact are interested in exactly the entire set of annotations that have been made on the annotatable items within their view. For instance, a viewer of a specific file would see exactly all of the annotations made to that file.

Two problems occur with this default behavior. The *clutter* problem occurs when an annotation tool presents users with too many annotations. The *delocalization* problem occurs when the user does not see pertinent annotations made on other parts of the artifact, or on related artifacts. In this thesis we examine the clutter and delocalization problems, describe a model of annotation to help us understand these problems, and discuss a software inspection tool incorporating ways to address these problems.

1.2 Terminology

Below, we briefly define some terms that we use when discussing annotated artifacts in this introduction. Chapter 3 more carefully defines the terms and explains the relationships between the concepts that these terms represent. Chapter 5 defines some of these terms mathematically as part of our mathematical model of annotation.

Item: An Item is a piece of information that may be annotated. An Item may either be divisible into other Items, or may be atomic. For instance, in natural-language text, a paragraph may be an Item divisible into words, which might themselves be atomic Items. Because we confer a special meaning on the term Item, we capitalize "Item" throughout this thesis.

View: A View is a set of Items that is presented to someone at one time. A View is determined by the Items it contains, not by the presentation of those Items. For instance, a Web page seen under two different browsers may look different, and users may choose to change parameters to alter its appearance, but so long as it contains the same Items, it is the same View. Because we confer a special meaning on the term View, we capitalize "View" throughout this thesis.

Viewer: A viewer is someone who accesses a View and its Items in read mode.

Annotator: An annotator is someone who annotates an Item. All annotators are also viewers, although the inverse does not hold.

Annotation: An annotation is a comment that an Annotator adds to an Item. An annotation is itself an Item, so an annotation can be annotated to create a threaded discussion.

Artifact: An artifact is an arbitrary set of one or more Views that are grouped together for convenience. Examples of artifacts include a requirements document or a source code file.

Filter: A filter is a mathematical function applied to a View that converts it into another (possibly different) View. Thus, when a viewer applies a filter to a view, that viewer sees a different set of Items (a different View) once the filter is applied. In this thesis, a viewer uses a filter to select which annotations to see when looking at a View.

1.3 "Basic" annotation fails to address interconnections

Basic annotation does not address the problem of delocalization. To illustrate our claim, we exhibit two simple annotated artifacts containing delocalized annotations. The first example is a famous short speech having nothing to do with software development, and the second is a simple piece of code with gross errors. Both examples are short, for ease of viewing. However, clutter and delocalization become greater problems with larger artifacts, which will have many more annotations and cannot be seen at a single glance or on one page. In all cases, we created annotations to clarify the delocalization issues, not because of any historical or technical interest.

1.3.1 Two Examples of Delocalization

Consider first Abraham Lincoln's Gettysburg address (Figure 1.2). There are three known drafts available to us: the Nicolay draft was a working draft (Library of Congress), the Bliss draft was the reading draft of the speech as given (Indiana University), and the Hay draft was written up by the author at a later date (Library of Congress). Assume that this speech is being studied in a class, and that students have annotated it. Let $\blacklozenge_n$ denote the beginning of the scope of the *n*-th annotation, and let $\otimes_n$ denote the end of the scope of the n-th annotation. The contents of some hypothetical annotations are listed on the page after the drafts.

Nicolay draft:

$\blacklozenge_1$ Four score and seven years ago our fathers brought forth, upon this continent, a new nation, conceived in liberty, and dedicated to the proposition that "all men are created equal"

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived, and so dedicated, can long endure. We are met on a great battle field of that war. We come to dedicate a portion of it, as a final resting place for those who died here, that the nation might live. This we may, in all propriety do. But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallow, this ground -- The brave men, living and dead, who struggled here, have hallowed it, far above our poor power to add or detract. The world will little note, nor long remember what we say here; while it can never forget what they did here.

It is rather for us, the living, we here be dedicated to the great task remaining before us -- that, from these honored dead we take increased devotion to that cause for which they here, gave the last full measure of devotion -- that we here highly resolve these dead shall not have died in vain; that the nation, shall have a new birth of freedom, and that government of the people by the people for the people, shall not perish from the earth. $\otimes_1$

Hay draft:

Four score and seven years ago our fathers brought forth, upon this continent, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived, and so dedicated, can long endure. We are met here on a great battlefield of that war. We have come to dedicate a portion of it as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

But in a larger sense we can not dedicate -- we can not consecrate -- we can not hallow this ground. The brave men, living and dead, who struggled, here, have consecrated it far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but can never forget what they did here. It is for us, the living, rather to be dedicated here to the unfinished work which they have, thus far, so nobly carried on. It is rather for us to be here dedicated to the great task remaining before us -- that from these honored dead we take increased devotion to that cause for which they here gave the last full measure of devotion -- that we here highly resolve that these dead shall not have died in vain; that this nation shall have a new birth of freedom; and that this government of the people, by the people, for the people, shall not perish from the earth.

Bliss draft:

♦$_2$ ♦$_3$ "Fourscore $\otimes_3$ and seven years ago our fathers brought forth on this continent a new nation, $\otimes_2$ conceived in liberty and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation or any nation so conceived and so dedicated can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting-place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

But in a larger sense, we cannot dedicate, we cannot consecrate, we cannot hallow this ground. The brave men, living and dead who struggled here have consecrated it far above our poor power to add or detract. The world will little note nor long remember what we say here, but it can never forget what they did here. It is for us the living rather to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us--that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion--that we here highly resolve that these dead shall not have died in vain, ♦$_4$ that this nation under God $\otimes_4$ shall have a new birth of freedom, and that government of the people, by the people, for the people shall not perish from the earth."

Figure 1.2: Lincoln's Gettysburg Address. Three versions of Lincoln's Gettysburg Address, side by side.

Annotations:

#1: I don't remember – could somebody tell me the order of these drafts? – *Student A.*

#2: What (if anything) should w e read into Lincoln's use of the masculine gender exclusively throughout this document? - *Student A.*

#3: "Fourscore" is tw o w ords, isn't it?". – *Student B.*

#4: I find it significant that "under God" appears only in the Bliss version, neither the version before nor after it….. – *Student C.*

The annotated speech indicates some features of basic annotation. Annotations #2 and #3 both begin at the beginning of the Bliss draft, but the scope of annotation #2 is clearly at least the first two phrases, and the scope of annotation #3 is only the first two words. Students in a history class would probably be interested in any or all annotations except #3. Conversely, probably only someone concerned with document reproduction, or possibly 19th-century grammar and usage, would be interested in annotation #3. Essentially, each of the viewers sees more annotations than they need to see. If there are large numbers of annotations, they would spend a lot of time wading through unnecessary annotations. They are confronting the *clutter* problem.

They also confront the *delocalization* problem. When dealing with material that has changed between two versions of a document, it is rather arbitrary as to which version gets an annotation discussing that change. People interested a given change should see the annotation in whichever version of the document they are viewing. So viewers of annotation #4 should see the annotation on the phrase "this nation under God" associated with the shorter phrase "this nation" in the other drafts. Yet traditionally, only viewers of the Bliss draft will see annotation #4. Similarly, annotation #1is a question about all three drafts. Anyone reading any draft should see it.

Annotation #3 brings up another problem of delocalization. The use of masculine nouns throughout the text is of interest wherever it occurs in each draft, not just at the beginning of the Bliss draft. So people looking at any part of any draft containing a masculine noun might be interested in this annotation and a discussion thread that might emanate from it.

Thus, we have taken three versions of a simple English-language text, and created hypothetical annotations on it which might be of interest not only to those reading the annotated version, but to those reading other versions, as well.

Now consider the following three source code functions. Again, the scope of the n-th annotation is denoted by $\blacklozenge_n$ at the beginning, and $\otimes_n$ at the end. The annotations indicate simple errors, some of which are catchable by a compiler.

Annotation #3 proposes a minor rewrite of function *c()*, and is not of interest outside of *c()*. However, it affects multiple lines in the function. If, in one of the statements not reproduced ("/* stmts. */"), the code uses *d*, it is important for a programmer to know that. In essence, the scope of annotation #3 is a set of statements of the function *c()*.

Annotation #2 deals with just one line of function *b()*. The scope of this annotation is the line being annotated, because that is the only affected portion of the code.

Annotation #1 is trickier. It states that the three functions a*()*, *b()*, and *c()* cannot ever work together because they are computed circularly. Although the annotation was placed on function *a()*, it could just as well have been placed on function *b()* or function *c()*. Annotation #1 ought to be visible to viewers of functions *b()* and *c()*, since people interested in the workings of functions *b()* and *c()* need to know something about the workings of function *a()*.

```
♦1 double a(int x) ⊗1
{
    int za;
    za = c(x) + 7;
    return za;
}
```

```
double b( int x2 )
{
    int z = 0;
/* stmts. */
    ♦2 z = a(z); ⊗2
    return z;
}
```

```
int c( int x3 )
{
    int d = 0;
    ♦3 double d; ⊗3
/* stmts. */
    d = b(zc);
    return (float)d;
}
```

Figure 1.3: Three interrelated functions.

Annotations:

#1: There is a circularity problem – a() calls c(), c() calls b(), and b() calls a().

#2: z is an int, but it is set equal to a function that returns a double.

#3: Instead of making d a double and casting it to a float w hen returned, just declare d as a float initially, and cast the assignment to “d = (float)b(zc)”.

The above problem is an example of Soloway’s original *delocalization* or *delocalized plan* problem, wherein ”pieces of code that are conceptually related are physically located in non-contiguous parts of the program” (Soloway et. al., 1988). This delocalization problem generalizes beyond code. For instance, in any document, it would generalize to repeated misspellings of words. Ideally, if one sees a misspelling one should only have to flag it once. This behavior, seen in some word processing tools, can be generalized beyond spelling. This is the same sort of delocalization problem as seen in the annotation concerning the gender-specificity of the language in the Gettysburg address. We call this delocalization between areas of a document or artifact at the same conceptual level *lateral delocalization*.

The delocalization problem generalizes beyond code to other software artifacts such as designs, requirements, and test plans (Stein et. al., 1998). For instance, suppose a requirement for some system states that a certain user screen must have a red background. The test plan may contain a test that passes if the background is red. A customer annotation of the specification requesting that the screen have a green background would be of interest to the test plan writers. As another example, consider a primary text and annotations made of it, such as the glossed bibles or Gettysburg address referred to earlier. Someone viewing a critique of such a text might be interested in annotations made on the primary source, as well as being interested in annotations made on the critique. These are instances of delocalization between two conceptually different documents (a specification and its test plan, a primary text and its critiques). We call this sort of delocalization *longitudinal delocalization*.

Annotation #4 on the Gettysburg Address exemplifies a third type of delocalization, *historical delocalization*, which occurs when annotations of one version of a document are of interest to viewers looking at other versions of the same document. As another example, annotations that a customer made to release 1 of some software might be interesting to the developers working on release 2. Or editors looking at a draft of a document might want to access their annotations on a previous draft in the context of the latest draft, to see if their concerns had been addressed.

1.3.2 Filtering to Address the Shortcomings of Basic Annotation

Filters can be used to address the clutter and delocalization problems.

Clutter has been addressed previously, by annotation filtering (Malone et. al., 1995 and 1997). In essence, some factor is used to separate out the annotations of interest to a given viewer. Typical examples of filters would be:

- Show all annotations by a given person or persons.
- Show all annotations made on or after a given date.
- Show all annotations of a given type, if the annotation system allows typed annotations, as do many software inspection systems (Knight and Myers, 1993; Mashayekhi, 1995).

We call these filters *narrowing filters* because they reduce the number of annotations seen. Thus, narrow-filtering alleviates the clutter problem.

To address the delocalization problem, we propose a new type of filtering that we call *broad-filtering*. The idea behind broad-filtering is to show a viewer who is looking at a View of an artifact not only those annotations made within that View, but also selected annotations made elsewhere in the artifact. Thus, viewers see a broader set of annotations than those on their present View. For instance, in the code example above, the viewers of all three functions, *a(), b(),* and *c()* would see annotation #1, because it affects them all.

Narrow-filtering is conceptually easy to deal with, and has been studied extensively. Broad-filtering is trickier to apply. To successfully apply broad-filtering, a system must be able to determine what annotations are likely to be of interest elsewhere in the artifact, and where those other places of interest for a given annotation might be.

In Chapter 3 we describe the delocalization problem in more detail, especially for software artifacts, to further motivate a need to address this delocalization problem. We point out that we must address the clutter problem at the same time to be able to effectively ameliorate delocalization. As part of our description, we develop the following hypotheses concerning delocalized annotation:

> ***Hypothesis 1:*** *Broad-filtering can be used to successfully ameliorate delocalization.*
>
> > ***Hypothesis 1.1:*** *Broad-filtering can be described mathematically in such a way that it can be applied clearly and unambiguously.*
> >
> > ***Hypothesis 1.2:*** *Broad-filtering can be implemented in an annotation tool.*
>
> ***Hypothesis 2:*** *Narrow-filtering can be successfully combined with broad-filtering to simultaneously reduce clutter and ameliorate delocalization.*
>
> > ***Hypothesis 2.1:*** *Simultaneous narrow-filtering and broad-filtering can be described mathematically in such a way that they can be applied concurrently without ambiguity.*
> >
> > ***Hypothesis 2.2:*** *Simultaneous narrow-filtering and broad-filtering can be implemented in an annotation tool.*

1.3.3 Annotation and Software Inspection

The primary motivation behind our interest in delocalization among annotations is the occurrence of delocalization in software inspection. Software inspection has been a useful technique for efficiently and cost-effectively finding defects in software since the 1970s (Fagan, 1976). Tool-supported software inspection came of age in the 1990s (MacDonald and Miller, 1999). In tool-supported inspections, faults are marked by annotations made to the inspection material in some way. We observe that all three types of delocalization among annotations – lateral, longitudinal, and historical - can occur during software inspection.

We have archival evidence of lateral delocalization, especially when dealing with the interface between two different pieces of software (Stein et. al., 1997). One instance of lateral delocalization we have seen occurs in design documents, where people find fault with an interface between two or more parts of the design. Some people attach the interface fault to one end of the interface, and others attach the same fault to the other end of the interface. Ideally, the interface itself would be annotatable, but even then there would be delocalization because viewers of Items at the ends of the interface might want to see the annotation. Another instance of lateral delocalization occurs with an annotation on a function call in a procedural language, or equivalently, a method invocation in an object-oriented (OO) language. Annotations made on the called function may be of interest to the callers of the function, and conversely.

One example of longitudinal delocalization, which we have seen in software inspections, occurs when someone indicates that the design cannot meet a requirement, perhaps a global system requirement concerning performance or reliability. An annotation indicating that a design *does not* meet a requirement may not be of interest to a requirements analyst. But an annotation indicating that a design *cannot* meet a requirement is clearly of interest to a requirements analyst and perhaps a customer, making it a longitudinally delocalized annotation.

Historical delocalization is intrinsic to software inspection because the inspected version of an artifact is not the same as the post-inspection version of that artifact, which contains the changes mandated by the inspection. Historical delocalization occurs with software inspection when a viewer wants to see the annotations made during an inspection in the context of the view of the system with defects fixed. We have seen evidence that annotations made during an inspection can contain important rationale for why certain decisions were made (Stein et. al., 1997), rationale that would be lost if these annotations were unavailable. And annotations from an old inspection would be hard to find were they not linked to the latest version of an artifact.

In Chapter 4, we propose modifications to the Humphrey protocol that we have used in that past for software inspection (Humphrey, 1989; Mashayekhi, 1995; Stein et. al., 1997) to address clutter and delocalization problems. We also develop the following research hypothesis concerning the delocalized annotations within software inspection.

> ***Hypothesis 3:*** *People can make use of an annotation tool containing narrow-filtering and broad-filtering for software inspection.*
>
> > ***Hypothesis 3.1:*** *Reviewers in a software inspection will find faults during the Discussion phase of inspection by seeing foreign annotations.*
> >
> > ***Hypothesis 3.2:*** *Reviewers in a software inspection will add to the discussion threads of foreign annotations.*

1.4 Mathematical Treatment of Clutter and Delocalization

As we began our research, we understood how to apply narrow-filtering to reduce clutter among annotations, but we did not know how to systematically apply broad-filtering to address delocalization. Thus, we also did not know how to integrate solutions to the clutter and delocalization problems, either conceptually or practically.

To better understand clutter and delocalization, and how to address these problems for annotations, we have developed a graph-theoretic model of annotation. This model treats the Items to be annotated as nodes in a directed graph, and connects these Items by relationships that form the edges of the graph. These relationships are typed, so that we can perform graph traversals. In our formalism, annotations are themselves Items, attached to the Item annotated by a relationship of type “annotates”.

Chapter 5 of this thesis contains this mathematical model of annotation, and proves some theorems which suggest that this model is a robust way to describe annotated artifacts. The model also shows us how to formally apply filters to an artifact to change the visibility of annotations within a View. The model incorporates composition of narrowing and broadening filters to achieve great flexibility in allowing a viewer to choose which annotations to associate with a given Item in an artifact.

To illustrate the use of our model, Chapter 6 presents five applications. The first application is filtering of a set of object-oriented (OO) software artifacts. Artifacts of this type showcase the power of filtered annotations in OO software development. Inspection of OO artifacts motivated our interest in the problem of delocalization. To show the generality of our model, our second example shows how to apply our model to a general textual document unrelated to software development, with minimal internal structure. The final three applications discuss existing tools for software inspection or collaborative authoring, and show how our model can explain their frameworks. By showing these examples, we are suggesting that our ideas could be applied to many existing software tools dealing with annotation.

In Chapters 5 and 6, we also discuss how certain theorems of our model and our ability to model various artifacts and the workings of various tools allow us to draw conclusions concerning the research hypotheses we developed in Chapter 3.

1.5 The AnnoSpec Tool

It is not enough to create a mathematical model of annotation, and to describe conceptually how it can be used. Beyond these, we must build a tool that incorporates our model of annotation, and use that tool for actual annotations of real software artifacts, in order to provide a “proof-of-concept” for our model of annotation. Therefore, we built the AnnoSpec annotation tool to *Anno*tate and in*Spec*t software artifacts. We designed AnnoSpec to support the inspection protocol we define in Chapter 4. We describe AnnoSpec in Chapter 7. As constituted at present, AnnoSpec has two significant limitations.

AnnoSpec only supports textual artifacts, making it suitable for code inspections, but unsuitable for inspections of diagrammatic specifications and designs. Additionally, AnnoSpec does not automatically determine the delocalization relationships. The person setting up an inspection (or informal annotation event) must determine beforehand the interconnections among Items, and manually enter these interconnections into the tool so that it can perform broad-filtering. These limitations constrain the type and size of artifacts that we can inspect. Ideally, AnnoSpec could parse the artifact and determine the delocalization relationships. For instance, it could parse code and determine that any annotation made on a call to a function *f()* should be visible to a viewer looking at *f()* or any other call to function *f()*.

Having built AnnoSpec, we could now see whether software professionals could use it to address delocalization problems that arise during software inspection. We analyzed eight software inspections using AnnoSpec, performed by 24 working software professionals pursuing a graduate degree in software engineering at our university. Chapter 8 reports the results of our pilot study, and discusses how those results confirm or rebut the research hypotheses concerning software inspection that we state in Chapter 4 of this thesis.

1.6 Research Contributions

This thesis makes three contributions to the state of the art of annotating software systems.

1. It taxonomizes ways in which delocalization occurs within annotated artifacts.
2. It develops a mathematical model of annotated artifacts that allows us to express annotation visibility along various axes.
3. It verifies that a system can be built to ameliorate delocalization, and can be used by software professionals in formal software inspection.

CHAPTER 2: RELATED WORK

Previous work related to this thesis falls into five general areas.

Delocalization and Related Problems: Previous work has identified the problem of delocalization and the related problem of traceability, and includes the development of systems to address these problems. Some of this work involves attempting to capture the rationale for decisions made concerning requirements and designs.

Information Handling: Our strategy for attacking clutter is related to work on information filtering, storage, and retrieval.

General Annotation: Annotation has also been a subject of previous study. Our work is related to a number of generalized annotation systems that have been built in recent years.

Collaborative Writing: Collaborative writing software often has an annotation component within it, and so forms a body of related work.

Software Inspection: Software inspection is the motivating problem for our work. Within software inspection, there are three areas of interest. Those areas are models of the software inspection process including the scope and stages of inspection, methods of reaching consensus asynchronously, and tools for computer-assisted software inspection.

2.1 Delocalization, Traceability, and Rationale

Soloway defined the delocalization ("delocalized plan") problem for code as occurring when "pieces of code that are conceptually related are physically located in non-contiguous parts of the program" (Soloway et. al., 1988). We call this lateral delocalization. Whorf, an early tool that addressed delocalization, identified conceptually related pieces of code and hyperlinked them together (Brade et. al., 1994). ICICLE, an early software inspection system, hyperlinked multiple occurrences of the same item together in C and C++ code, serving the same purpose but with less generality and correspondingly more power (Brothers et. al., 1990). Modern development tools also incorporate the ability to find various occurrences of an item (Visual Studio).

A problem related to delocalization is the well-known traceability problem. In its most common form, the problem is to trace the requirements for a system to and from the code that implements the requirements, ideally tracing through the design of the system to do so. Many tools have been developed to aid traceability. Experimental tools have included PRO-ART (Pohl, 1996), READS (Smith, 1992), and SCIDS (Lees and Jenkins, 1995), and EColobar (Takahashi et. al., 1996). Traceability has matured to the point that commercial tools are available to support traceability as part of a software development environment, such as Razor (Visible Corp.) and RequisitePro (Rational). Traceability is essentially a problem of longitudinal delocalization, in which the delocalized information is spread among the requirements, design, and code of a system. Corriveau states that the best way to address the traceability problem is through hyperlinkage (Corriveau and Hayashi, 1994).

One aspect of traceability that is often overlooked is the importance of “pre-requirements” traceability – finding out where requirements came from (Pohl, 1996). It is recognized that to address traceability, especially pre-requirements traceability, it is important to know who placed what requirements on a system (Gotel and Finkelstein, 1994; 1995; 1996).

Related to the question of who placed requirements on a system, or who made a design decision, is the question of why such a decision was made in the first place. This is the problem of capturing rationale for decisions. Capture of rationale is especially important for software maintenance, where the people who made the original decisions may be long gone from the project or the organization. Their departure is a form of organizational memory loss (Walsh and Ungson, 1991). Lougher and Rodden have developed groupware to encourage the capture of rationale for software maintenance (Lougher and Rodden, 1993a,b). They are essentially addressing a problem of historical delocalization. Their model allows flexible typing and scoping of annotations.

We extend the solutions proposed for the above delocalization problems by linking annotations on various parts of the artifact instead of linking parts of the artifact themselves. Thus our approach is complementary to the more common approaches to the above delocalization problems. In our model and tool, an annotation made on a related artifact can be filtered to the viewer without their having to look at all related artifacts. This is especially useful for making the crucial link between an item and the person responsible for that item. We have observed situations in which rationale for decisions has been explained during inspections (Stein et. al., 1997). We propose that making visible historically delocalized annotations can allow that rationale to be captured and made part of the development history.

2.2 Information Handling

This thesis considers the relationship between an artifact and its annotations. Both the artifact and the annotations are pieces of information. We have both conceptual and practical challenges in filtering this information to address clutter and delocalization. We also have practical problems in storing and retrieving this information.

2.2.1 Information Filtering

Traditional information filtering is concerned with ways to address what we call the clutter problem. The simplest way to filter is to narrow-filter information by known parameters, such as date of creation or author. This is a staple activity within database systems (Date, 1977).

Information filtering systems may go beyond narrow-filtering, to allow users to significantly tailor the system to their needs via simple end-user programming. Examples of such systems are Oval (Malone et. al., 1995) and Information Lens (Malone et. al., 1997).

One important use for information filtering is to manipulate the large repositories of information that constitute “organizational memory”. Walsh, et. al. discuss organizational memory in detail (Walsh and Ungson, 1991).

We have created a graph-theoretic model of our annotation system, incorporating its information filtering attributes. Some other research also involves the modeling of coordination tasks. This research includes the Trellis model (Furata and Stotts, 1994) and Rodden’s model of awareness for cooperative applications (Rodden, 1996). Our model has a narrower scope than these models, and is correspondingly more detailed.

2.2.2 Information Storage and Retrieval

Our model stores each piece of information with links to other information. In a real system, we need a way to store and retrieve these data, which are largely in the form of interrelated objects. GRAS (Kiesel et. al., 1993) and PLEIADES (Tarr and Clarke, 1993) are object management systems that store objects in ways that make use of their relationships to other objects. They form a framework for object storage that could be used to store our Items and their annotations. However, they do not deal specifically with annotations or their information filtering implications. Although we use similar ideas to those used by these tools for object storage and retrieval, we use the commercial object-oriented database PSE (Object Design) for storage. Our ideas on storing objects were also influenced by Goeschka, et. al. (1998).

2.3 General Annotation Systems

Work in general annotation systems is relevant to our work because we are modeling and building an annotation system. Although ours is a special-purpose annotation system, our work may be informed by studying general-purpose systems.

2.3.1 Ovsiannikov's Theory of Annotations and Non-local Referencing

Ovsiannikov, et. al. (1999) have compared seventeen annotation tools, including their own general-purpose Web annotation tool Annotator. They recognize the delocalization problem for annotations, which they call the "non-local referencing" problem. They do not taxonomize the problem, as we do, but they consider it in the broader scope of annotation systems in general, while we specialize our interest to software systems. They explicitly limit their discussion to annotation systems not used for what they call "sharing", a category that includes inspections; yet many of their ideas and results are of interest to us.

Some properties that they claim are desirable in annotations systems, based on user surveys they have conducted, apply to systems used for inspections and general sharing. Those properties are:

1. Ability to annotate pictures
2. Ability to search annotations by keyword
3. Ability to use both pen and keyboard/mouse input for making annotations.
4. Ability to mark a document as a paper document would be marked, for instance by writing in margins, highlighting portions of a document, and crossing out sections of a document.
5. Support for non-local referencing.
6. Ability to view annotations without changing the formatting of the document.

Unlike Ovsiannikov, we focus on "sharing" systems, especially systems for annotating and inspecting software artifacts. Our work provides a model for supporting non-local referencing (#5) in general annotation systems. By limiting ourselves to textual systems, our tool preserves document formatting (#6). The other four items, especially the first two, would be useful for collaborative annotation and inspection.

2.3.2 General Annotation Tools

Ovsiannikov et. al. created a Java-based tool called the Annotator with the above properties. They did not discuss the underlying model for their way to address non-local referencing, but they discuss their tool in enough detail that we can compare it with AnnoSpec. Some major points of difference are:

- They identify an annotation with a "clump" of Items, instead of associating it with one Item as a native annotation and then making it a foreign annotation on other, related Items.
- They used a relational database, allowing them to easily do keyword searches.

- The Annotator runs as an extension of Netscape Composer, so it is platform-specific.

CritLink (Yee, 1998) is a Web-based annotation tool. Basically, one puts a document to be annotated onto the Web using CritLink, and people can annotate a portion of the document by highlighting that portion they wish to annotate. Annotations show up as inline flags, one marking each end of the annotated information. Thus CritLink is a very flexible, general-purpose annotation tool. In our experience, having tried it numerous times, it is also very slow.

Third Voice (Third Voice) is a Web-based annotation tool that is similar to CritLink, except that it only places a marker at the beginning of annotated text. Unlike CritLink, it is also fast enough to be usable, and it allows for various levels of annotation visibility (annotator only, pre-defined group, world). But it sometimes radically alters the appearance of the underlying web page.

Remark is a tool for annotating documents in the PDF format (Remark). It allows users to annotate PDF documents as an overlay, with text highlighting and strikethroughs, and the ability to attach text, pictures, and audio annotations that become part of the original document.

DynaText is a commercial document publishing system that allows annotations (Dynatext). One feature of DynaText that addresses delocalization is that it allows users to hyperlink together multiple occurrences of the same Item.

Inote is a tool for annotating graphics (Bingler et. al.). It allows an annotator to attach textual annotations to various regions in an image and to store the annotations in an associated text file which can be called up when the image is viewed.

Cubus ReviewIt is a tool for annotating text and images (Cubus Corp.). It permits users to annotate portions of images by drawing ellipses or rectangles within pictures and annotating the area within the closed curve. However, it treats the image as a whole, and does not handle Items within images. It also supports roles for annotation sessions, such as Reviewer and Author; however, it lacks any tie-in with a formal software inspection protocol.

An active area of investigation for modern annotation tools is multimedia annotations. DIANE is a well-known multimedia annotation system (Bessler et. al., 1997). It allows audio, video, and textual annotations to be made to multimedia documents, and allows for discussion threads by permitting annotations of the annotations themselves. The type of Item being annotated and the type of permissible annotation are not constrained (thus one could make an audio annotation of a video clip, for instance). DIANE does not filter annotations.

At the other end of the spectrum, basic annotation can be accomplished effectively without any special tools at all, merely by using e-mail (Diaper and Beer, 1995).

Neuwirth, et. al. (1998) have proposed six general capabilities that should be provided by asynchronous communication tools. Our AnnoSpec tool addresses two of these that existing inspection tools do not appear to address: we have a "task-tailorable" representation, and we implement "asynchronous awareness". We discuss these further in Chapter 7. Additionally, unlike all tools except the Annotator, AnnoSpec supports broad-filtering to address delocalization. DynaText allows broad-filtering, but it must be done manually. However, AnnoSpec only handles text, unlike most of the above tools.

2.4 Collaborative Annotation in Authoring Tools

Collaborative authoring tools often contain support for annotations. These are general-purpose tools that can be used for a variety of documents, and are not limited to, or even designed for, writing and analyzing software. Each tool meets some of our requirements for an improved collaborative annotation system, but no tool meets all the requirements. Some tools that support various filtering activities are described below.

The PREP editor supports editing a document in a multi-column layout, in which one column is the document and every other column is a subset of annotations, filtered by author

(Cavalier et. al., 1990). This elegantly supports filtering on one dimension. PREP has been shown to be slightly less effective for collaborative writing than face-to-face (FtF) collaboration (van der Gerst and Remmers, 1994). Part of the problem is the difficulty in handling workflows and merging different copies of the same part of the document (Tammaro et. al., 1995). Tammaro also found that for simple document types, the annotation capabilities of PREP were not needed. Rather, a set of “canned” annotation types, combined with robust support for merging multiple versions of a section and managing work flow, were satisfactory.

The SEPIA cooperative hypermedia authoring environment allows collaborative editing of hyperdocuments. It includes an "argumentation space" for threaded discussions on the authored material. This space is reminiscent of the decision support subsystem of an automated software inspection system (Streitz, et. al., 1992).

MILO is a collaborative authoring tool that permits all annotations in a document to be accessible from any view of the document, via a “note space” (Jones, 1995). But it doesn’t filter the annotations by their relationship to the material being shown.

SHADOW is a collaborative authoring tool for textual documents that uses the concept of electronically pasting information over text (Pino, 1996). It is designed to allow viewers to see multiple versions of text. When used to paste annotations over text, it implicitly incorporates annotation scoping. Increasing the scope of an annotation approximates broadening.

Collaborwriter is a collaborative writing tool that permits annotations of different types (McAlpine and Golder, 1994). Users are able to perform type-specific annotation behavior, including changing the scope of some types of annotation.

Our AnnoSpec tool is not an authoring tool, in that it is not integrated with any other authoring functionality, such as editing a document within the tool. However, the ideas underlying each of the above tools provided some motivation for the functionality of AnnoSpec, or motivated some ideas in our model of annotation, such as annotation scoping and typing, and inclusion of threaded discussions with annotations.

2.5 Software Inspection

Software inspection is a common software development activity that involves significant annotation of software artifacts. Inspection is done on a range of artifacts, from system specifications through designs and implementations, as well as on related artifacts like test plans and user manuals. Software inspection was the motivating problem behind this thesis.

There is a rich literature on software inspection, focusing on two interrelated facets: the protocol for software inspection, and the set of tools for computer-assisted software inspection. Although we treat these facets separately, there is a strong relationship because the tools are often designed to support a specific protocol.

2.5.1 Inspection Protocols

2.5.1.1 Time-Space Taxonomy of Inspection Styles

Mashayekhi (1995) taxonomized the three ways to hold a software inspection in space and time. The choice of which to use constrains the choice of tools that will be used with the inspection.

1. *Face-to-Face (FtF) Inspection.* In "traditional" FtF inspections, the participants in the inspection process meet together in the same place at the same time for an inspection meeting. Depending on the protocol used, this meeting may be preceded and/or followed by the participants working individually. But the defining characteristic of an FtF inspection is its reliance on a meeting of all participants as a crucial part of the inspection. FtF inspection can be performed with pencil and paper, with no tool support at all. However, automated systems can make some aspects of inspection easier.

2. *Distributed, Synchronous Inspection.* A distributed, synchronous inspection is similar to a traditional FtF inspection, except that the participants are distributed in space at the time of the meeting. This requires that the participants have various tools at their disposal to encourage synchronous communication, such as an audioconferencing setup. Also, some form of tool support for handling inspection materials is a virtual necessity.

3. *Distributed, Asynchronous Inspection.* During the late 1980s and early 1990s, advances in distributed systems, networks, and user interface technology enabled distributed, asynchronous meetings to become a viable alternative to FtF meetings (Bly et. al., 1993). This idea is appealing for software inspection for two reasons. First, it is not always

possible to get the desired parties in the same place at the same time. Even if an audio/video conference is possible, people who are geographically distributed among the various longitudes around the globe may not be able to find a mutually convenient meeting time (Stein et. al., 1997). Even when people are collocated, it has also been discovered that simply finding a time when all the right people are available for an inspection meeting can take weeks (Votta, 1993; Ballman and Votta, 1994). A distributed, asynchronous inspection does away with the synchronous meeting altogether. Instead, participants access the inspection materials at times of their choosing, and work together asynchronously to perform all stages of the inspection. Distributed, asynchronous inspection has the most stringent tool requirements of any form of inspection.

2.5.1.2 Fagan Inspection

Fagan created the idea of formal software inspection at IBM in the 1970s (Fagan, 1976). His methodology is still followed today for many FtF inspections (Berman, 1998), and serves as the basis for later protocols, including some for distributed inspection (synchronous and asynchronous). In this thesis we describe Fagan's protocol in detail, and refer to later protocols by their differences from Fagan inspection.

Fagan inspection assigns participants in the inspection to various roles. The *Author (*or *Producer)* is the author of the inspected material. The *Moderator* is the person in charge of the inspection. The *Recorder* records defects brought up in the inspection meeting. *Reviewers* inspect the material to look for defects. The Moderator and Recorder double as Reviewers.

Fagan inspection proceeds in five phases:

1. *Kickoff.* The participants get together and the Moderator hands out the inspection material. The Moderator or Author may give an overview of the material at this time, and place it in the context of the project. A kickoff meeting is optional.

2. *Preparation.* The Reviewers each look over the inspection material individually, and note any defects they find.

3. *Inspection Meeting.* The participants have an FtF meeting, at which they discuss the defects they have found. The Recorder writes down the defects. The purpose of the inspection meeting is to identify faults, not to resolve them. Thus, discussion of how to fix defects found is prohibited, because of a perceived tendency to sidetrack the meeting.

4. *Rework.* The Author fixes the defects found, consulting with members of the inspection team or others privately as desired. After rework is finished, the Moderator checks the modified material, and certifies the inspection as complete.

We wish to bring out two other points about Fagan inspection that are salient to future modifications.

- Fagan inspection was initially designed to inspect code, not upstream documents such as specifications or designs.

- Large volumes of code are divided into many inspection groupings, with one rule for division being that an inspection meeting should only cover as much code as can be reasonably inspected in two hours. Exactly how much material this is, and how it relates to artifacts other than code, has been an active area of research for the past 20 years (Christenson, 1993; Gilb and Graham, 1993; Gilb et. al., 1998).

2.5.1.3 Humphrey Inspection

One early modification to Fagan inspection was made by Humphrey (Humphrey, 1989). Our research group has historically used Humphrey inspection as the basis for our work. Humphrey has the same Kickoff and Preparation phases as Fagan, after which the protocols diverge.

Before the inspection meeting, the Author collects individual defect lists from each Reviewer, and merges them all into a master defect list for the inspection, eliminating any duplicates or non-errors. This phase of inspection is called *Fault Correlation*.

In the inspection meeting, each participant gets the master list of defects. The purposes of the meeting are both to verify that the proposed errors are truly errors, and to resolve the errors when possible. By "resolve", we mean to determine the parameters of the solution, not necessarily work out the details. Resolving faults in the inspection meeting is a fundamental departure from Fagan's protocol. The inspection meeting is sometimes referred to as a *Discussion* or *Fault Resolution* meeting.

Finally, after the rework is done, the Moderator reviews the reworked artifact and verifies completion of the inspection.

Our research group has used Humphrey's model as the basis of our work on distributed inspection, both synchronous and asynchronous (Mashayekhi et. al., 1993, 1994, 1996; Stein et. al., 1997). Our one modification, also suggested by Murphy and Miller (1997), has been to have the Moderator, not the Author, perform Fault Correlation. This reduces the potential for conflict of interest, and seems in line with the overall duties of the Moderator. One main reason we chose Humphrey's protocol was that the use of a defect list helps to structure a distributed meeting (Mashayekhi, 1995). Since then, we have observed other advantages of this protocol.

First, it ensures that the solution to the problem, and discussions leading to that solution, are seen by the Reviewers. In Fagan inspection, an Author need not formally clear corrections with anyone but the Moderator. When Fault Resolution is part of the inspection meeting, all the interested parties may get involved in the process.

Also, much discussion of rationale and design alternatives comes out in such discussions. This is valuable information to save for future reference, as people may seek to discover the rationale behind various decisions that were made.

In Chapter 4, we discuss modifications we have made to the inspection protocol that relate to annotation filtering.

2.5.1.4 Other Inspection Protocols

The other inspection protocols that we describe are all modifications of Fagan's or Humphrey's protocol.

One series of modifications has been the classification of errors by type. Many practitioners have classified errors by type, and have developed checklists for people to use in looking for errors (Baldwin, 1992; Nyquist and Henricson, 1992; Chernak, 1996). Parnas and Weiss (1985) proposed "Active Design Reviews". Instead of breaking up a large document into sections and inspecting each section separately, they proposed breaking up the inspection into multiple smaller inspections, each looking for different types of errors. Knight and Myers proposed "phased inspection", replacing an inspection meeting by a number of rigorously-defined single-inspector and small multiple-inspector phases, each concerned with a specific property or properties of the artifact (Knight and Myers, 1993).

Johnson, et. al. (Johnson and Tjahjono, 1993; Johnson et. al. 1993; Johnson, 1994, 1996) have developed the FTArm protocol for distributed, asynchronous software inspection. This protocol is similar to Humphrey's, except without Fault Correlation. Because it is designed to be used for distributed asynchronous inspection with tool support, it utilizes voting on proposed solutions developed in the Fault Resolution phase for resolving issues. Issues on which there is no agreement are set aside for a final group meeting, either FtF or distributed.

Tervonen has proposed the GRCM (Goal-Rule-Checklist-Metric) model for software quality, and has applied this to inspection (Tervonen, 1996). GRCM inspection is basically another modification of Humphrey inspection, one that makes heavy use of metrics in conjunction with checklists. For instance, in an inspection of OO classes, checklists would contain metrics to be used to decide the quality of the class, such as how many attributes or methods it had. Additionally, GRCM involves small reviews by subgroups of the inspection team during the Preparation phase, as well as containing a whole-team Fault Resolution meeting.

Tom Gilb has recently argued that software inspections are more useful for determining overall process and product quality than for finding defects (Gilb et. al., 1998; Gilb, 1999). As such, he advocates intensive checking of a portion of an artifact, including standard re-reviews, to arrive at an estimate of defect density for the overall product. Then, based on the defects found during inspection and historical data, one further step of inspection, also accepted by Tervonen, et. al.(1999) is to estimate the defect density remaining within the artifact after inspection.

2.6 Software Inspection Tools

Many tools have been developed to partially or fully support various inspection protocols. MacDonald and Miller have twice published overviews of such tools (MacDonald et. al., 1995; MacDonald and Miller, 1999). We discuss a few of the tools that are relevant for our work.

2.6.1 Synchronous Inspection Systems

The first level of support for software inspection is tool support for inspection meetings, whether FtF or distributed synchronous inspections. Such a tool may require all participants to be in the same specially-equipped room, or it may support distributed inspection, allowing people to converse via audio-conference and/or video-conference.

ICICLE (Brothers et. al., 1990) was built to support the tasks performed during synchronous inspection of C and C++ code. ICICLE assists individual users in the comment-preparation phase of code inspection. It provides an environment for a synchronous inspection meeting, and uses computer support to permit a paperless meeting. ICICLE uses its knowledge of C and C++ to address delocalization within the code; someone looking at a variable or function can use the tool to find other uses of that variable, or other calls to that function.

Collaborative Inspection Agent (CIA) is a document inspection tool (Gintell and Memmi, 1992). CIA supports synchronous inspection of all work products at various stages of the life cycle. It supports collaborative work by simultaneously displaying information on multiple users' screens, and allowing participants to play inspection roles.

Scrutiny (Gintell et. al., 1991) is a collaborative inspection system that has been successfully used for professional software development. It supports reviewers in a synchronous meeting, following a protocol that is essentially similar to that of Fagan.

2.6.2 Asynchronous Inspection Systems

InspeQ is a tool that supports Knight's phased inspection protocol (Knight and Myers, 1993). By its nature, the InspeQ tool is asynchronous because it does not involve any meeting support. The phased inspection process itself, however, involves a final meeting to review the results of the various inspection phases.

hyperCode is a Web-based tool used to support distributed, asynchronous inspection (Perpich et. al., 1997). This tool supports textual inspection of line-oriented artifacts, in a modified Fagan-style inspection. Reviewers annotate the artifact, and the Moderator and Producer collect the annotations and resolve the issues offline.

The Collaborative Software Review System (CSRS) uses the FTArm protocol (Johnson and Tjahjono, 1993; Johnson et. al., 1993). One key facet of CSRS is that it includes a process modeling language that can be used to program CSRS to support a wide range of inspection protocols, both synchronous and asynchronous. The system is implemented on top of EGRET, a multi-user, distributed, hypertext environment for asynchronous collaboration. CSRS/FTArm has been used to obtain statistically significant results about the effectiveness of software inspection in general (Johnson, 1997), and in particular to observe that the cost-effectiveness of asynchronous inspection is comparable to that of synchronous inspection (Tjahjono, 1995; Johnson and Tjahjono, 1998).

MacDonald, et. al. have developed the ASSIST tool, which is designed to be flexible with regard to inspection protocol and material (Murphy and Miller, 1997; MacDonald and Miller, 1999). Like CSRS, it contains a process modeling language that can be used to customize ASSIST to support different processes, and can also be used to enable inspection of graphical artifacts. ASSIST also contains support for checklist definition and integration into the inspection process.

Tervonen has recently developed the Web inspection Tool (WiT) for inspecting documents over the Internet. It marks issues by drawing vertical lines to the left of the areas on a Web page that constitute the scope of a comment. It is not clear from the literature whether the scopes of comments are pre-selected per inspection or not (Tervonen et. al., 1999).

SDT Corp. has a commercial tool called ReviewPro (SDT) that appears to follow Humphrey's protocol, with a metalanguage to allow companies to modify the process, and to enhance checklist and metric support. However, ReviewPro does not appear to allow participants to view the document and its annotations in the same window, or to associate annotations visually with specific parts of the document.

Our research group has built and experimented with various prototype tools for collaborative inspection. The Collaborative Software Inspector (CSI) provided computer support for synchronous inspection (Mashayekhi et. al., 1993). CSI used the Suite infrastructure (Dewan and Choudhury, 1994) The Collaborative Asynchronous Inspector of Software (CAIS) extended CSI to support distributed, asynchronous inspection of text (Mashayekhi et. al., 1994). CAIS was implemented on Suite and also on Lotus Notes (Thompson and Riedl, 1995). The Asynchronous Inspector of Software Artifacts (AISA) was a Web-based tool for distributed, asynchronous inspection of textual and graphical artifacts (Mashayekhi et. al., 1996). AISA has been used successfully for inspections in industrial settings (Stein et. al., 1997).

2.6.2.1 Decision Support in Asynchronous Software Inspection

In Chapter 4, we discuss decision support within our software inspection protocol. The issue of fault resolution in asynchronous Humphrey-style inspections has received some study. The question is how to determine if a fault has been resolved in an asynchronous inspection. Our tools CAIS and AISA, as well as WiT and CSRS, support a voting protocol for fault resolution. In this protocol, all inspection participants (except possibly the author of the artifact) must vote on a proposed fault resolution. We have found in field studies that participants disliked having to vote (Stein et. al., 1997), largely because they had to vote on every issue before it could be resolved, even though most participants were only interested in a small fraction of the issues. Part of the problem was that even abstaining from voting on an issue required work. Tan et. al. (1995) also point out that voting is not a part of most corporate cultures.

Other tools do not have any formal protocol for fault resolution. They just allow discussions to continue and make it a judgement call on someone's part as to when issues are resolved and what the resolutions are.

2.6.3 Inspection Tool Summary

All of the above systems either were the first to extend software inspection in some direction, or still have unique properties. Our support for delocalized annotations is a logical step beyond ICICLE's allowing inspectors interested in a certain C/C++ entity (such as a variable) to see all uses of that variable; we allow reviewers to see all annotations made on any annotatable Item. Support for delocalized annotations is our main advance in the state of the art of software inspection systems.

None of the inspection systems above broad-filter annotations. Many allow filtering of annotations by author or date, thus supporting narrow-filtering. An interesting aside is that the CSRS tool treats the artifact as a set of linked nodes, as we do, and considers annotations to be other nodes linked to the nodes of the artifact. They use this structure to allow people browsing the artifact to move back and forth among related nodes (Johnson et. al., 1993). However, they do not extend this linking to annotations, as we do. Nor do they develop a formal model of annotation.

CHAPTER 3: CLUTTER AND DELOCALIZATION PROBLEMS

In Chapter 1 we outlined the problems with basic annotation and our approach to those problems. This chapter describes the clutter and delocalization problems in detail, describes our proposals for addressing these problems, and lays out research hypotheses concerning these problems. Before describing this information in detail, we give more technically precise definitions of some of the terms that we defined in Chapter 1.

3.1 Concepts Related to Annotation

We use some common, natural-language terms in very specific ways throughout this thesis. This section discusses the concepts underlying those terms in detail.

3.1.1 Concept Definitions

3.1.1.1 Item

An *Item* is a piece of information that may be annotated. An Item may be composed of other Items, or may be atomic. The atomicity of an Item is a function of the model of the artifact and the annotation system. For instance, consider general natural-language text. Such text contains letters and other symbols, combined into words, further combined into paragraphs. In some circumstances, an individual letter or symbol might be an atomic Item. In other cases, a word might be an atomic Item.

An annotation itself is an Item, because it can be annotated to create or add to a discussion thread.

In our previous work and in the AnnoSpec documentation, we refer to an Item as a *Unit of Annotation (UA)*. Given our specialized use of the term Item, we capitalize it throughout this thesis.

3.1.1.2 View

A *View* is a set of Items that is presented to someone at one time. A View is determined by the Items it contains, not by the presentation of those Items. For instance, a Web page may look different when viewed with different browsers, and users may choose to change parameters to alter the appearance of the page. Yet so long as the page (however manipulated) contains the same Items, it is the same View. Other common Views include a page of a hard-copy document or a file of source code. For some types of Views, Items may be presented in time rather than in space. For instance, a file of audio or video information would also constitute a View, but this View would only be retrievable by sequential access, not by random access.

Given our specialized use of the term View, we capitalize it throughout this thesis.

3.1.1.3 Viewer

A *viewer* is someone who accesses a View and its Items in read mode. Thus a viewer is a member of the target audience of the annotators of Items in a View.

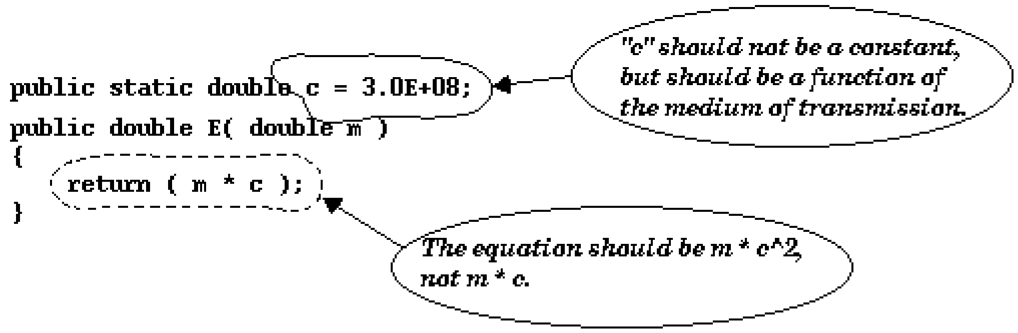
3.1.1.4 Annotator

An *annotator* is someone who annotates an Item. Annotators must be viewers of a View in order to be able to see the Items on which they may place annotations. The converse is not true. Specifically, annotators have read-write access to a View and its constituent Items, while viewers need only have read access. For instance, in the software inspection protocol we discuss in Chapter 4, the authors of an artifact can read the annotations that others make on it, but cannot annotate the artifact themselves – they are viewers who are not annotators.

3.1.1.5 Annotation

An *annotation* is a comment made on an Item by an annotator. That annotated Item is the *scope* of the annotation. An annotation is made on a specific Item *I*, not on Items that contain or are contained by I. For instance, suppose an annotation A is made on a paragraph of a document. Annotation A annotates the Item that is the paragraph, but it does not annotate any word Item contained within the paragraph, nor does it annotate the chapter Item containing the annotated paragraph.

We classify annotations into two types, depending on how they are presented to a viewer. A *native* annotation is an annotation presented to a viewer in association with the Item it annotates. This is the standard way of presenting annotations. A *foreign* annotation is an annotation that is presented to a viewer in association with some Item other than the Item annotated. (See Figure 3.1)



```
public static double c = 3.0E+08;
public double E( double m )
{
   return ( m * c );
}
```

"c" should not be a constant, but should be a function of the medium of transmission.
The equation should be m * c^2, not m * c.

Figure 3.1: Native and foreign annotations. This figure shows a very short Java method that contains two annotations. The lower annotation is a native annotation, indicating that the statement being annotated is incorrect. The upper annotation is a foreign annotation that was placed on the global parameter *c*, stating that *c* is not constant. The writer of method *E()* might be interested in knowing that a constant might be replaced by a method, even though the annotation suggesting this possibility was not made on the method *E()*.

3.1.1.6 Artifact

An *artifact* is an arbitrary set of Views grouped together for convenience. We use the term artifact in a very general manner in this thesis, as a convenient way to refer to a set of Views that people tend to think of as closely related. An artifact may contain text, graphics, and/or multimedia information such as audio or video. Although we do not address multimedia artifacts directly in this thesis, our ideas appear to apply to such artifacts. The only requirement we place on an artifact is that it be human-viewable. (For audio artifacts, this would be human-audible.) We call the set of all artifacts in a system its *artifacture.*

This thesis concentrates on annotation of software artifacts. There are many types of software artifacts. Some common ones are:

1. System specification
2. High-level design documentation, such as a class diagram of an OO system
3. Low-level design documentation, such as pseudo-code
4. Project plan
5. Source code
6. Test plan
7. User manual

These artifacts can take various forms. For instance, a specification may be written in natural-language text, in a graphical language such as the Unified Modeling Language (UML)(Fowler, 1997), or in a formal language such as RSML(Leveson, et. al., 1994). There is a similarly wide range of choices for the other types of documents listed.

We define "artifact" to give us a convenient shorthand to refer to one set of documentation. For instance, the set of material inspected in one software inspection might be considered an artifact. If a subset of that material is later inspected again, that subset would also be an artifact. Similarly, this thesis was first reviewed chapter by chapter, with each chapter considered to be a separate artifact. Later, in a final review stage, the entire thesis was treated as a single artifact.

3.1.1.7 Filter

A *filter* is a mathematical function that a viewer applies to a View to convert it into another (possibly different) View. Thus, when a viewer applies a filter to a view, that viewer sees a possibly different set of Items (a different View) once the filter has been applied. In this thesis, a viewer uses a filter to select which annotations to see when looking at a View. Two or more filters may be composed by piping the output of one filter to the input of the next to create the output View that a viewer will see.

One dimension along which to classify filters is by their effect on the number of Items in a View. A filter can either reduce or increase the number of Items seen.[1]

A filter can reduce the number of Items seen. Such a filter is called a *narrowing filter*, and the process of applying such a filter is called *narrow-filtering* or *narrowing*.

A filter can also increase the number of annotations that a viewer sees by showing annotations that were made on Items outside the present View. For instance, consider a View of the definition of an OO class *C*. A filter might be defined to show annotations that were made on a subclass of class *C*. Such a filter is called a *broadening filter*, and the process of applying such a Filter is called *broad-filtering* or *broadening*.

3.1.2 Relationships Among the Defined Concepts

We define a View as the set of Items it contains. But what does it mean for an Item to be visible, to be contained within a View?

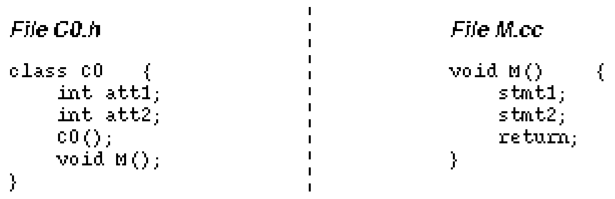
Consider a C++-language class *C0*, which contains a method *M()*, as shown in Figure 3.2. Suppose the class is defined in header file *C0.h* (call this View *C0.h*), which must contain the signature for method *M()*. Suppose that the code for the method is given in a separate source code file *M.cc* (call this View *M.cc*). In any development system, a viewer could use a browser or editor to see these two files separately. Thus it is reasonable to consider each of these

[1] Traditionally, the term "Filter" is applied to phenomena akin to narrowing filters. Our use of the term for broadening filters is somewhat unusual. We considered use of a term such as "broadener" or "expander". But eventually we will compose narrowing and broadening filters together, at which time we would need to invent a third new term. We felt it was simpler to use the term "filter" in a wider context than that in which it is normally used.

files to be a separate View. But the separation of class *C0* into two files gives rise to the following visibility questions.

1. View *M.cc* clearly contains method *M()*. Does View *C0.h* contain method *M()*?

2. Which of the above Views contains class *C0*?



File C0.h

```
class C0    {
    int att1;
    int att2;
    C0();
    void M();
}
```

File M.cc

```
void M()    {
    stmt1;
    stmt2;
    return;
}
```

Figure 3.2. Class C0 and its method M(). This figure shows header file *C0.h* and source code file *M.cc*. Each is a separate View. Which View(s) contain *M()*? Which View(s) contain *C0*?

Neither View contains all of class *C0*, because *C0* contains the definitions and implementations of all its attributes and methods, and neither file contains all of this. Yet an annotator must be able to annotate the class as a whole – to make an annotation of class scope – from somewhere. Thus at least one of the above Views must contain class *C0*.

View *M.cc* clearly contains the method Item *M()* because the entirety of the method is visible in that View. But *C0.h* contains the signature of *M()*, and possibly some documentation of it as well. It seems that one ought to be able to annotate the method as a whole from *C0.h*, because header *C0.h* contains enough method information that someone might annotate method *M()* to describe a high-level problem with it. Yet we must not allow people to annotate Items from Views in which they are invisible, because they do not have enough information about that Item in such a View to make annotation sensible.

In Chapter 5, we formalize the idea of the *kernel* of an Item, that set of sub-Items of an Item such that if the kernel is visible, the Item is visible. For our practical problem here, we would say that the kernel of a method is that method's signature, so method *M()* is visible from both file *C0.h* and file *M*.cc. We would also say that the kernel of class *C0* is the set of its attributes and methods, so *C0* is visible within file *C0.h*.

3.2 Clutter

Clutter occurs within an annotated artifact when the sheer volume of annotations makes it difficult to read the artifact and/or its annotations. Clutter may occur when an artifact is presented together with many annotations. This is the strategy followed on hard-copy artifacts in which an annotator makes annotations via margin notes. It is also used in tools such as PREP (Cavalier et. al., 1990) and SHADOW (Pino, 1996). With this strategy, a large number of annotations can make it difficult to read the artifact and/or annotations.

Alternatively, annotations and the unannotated artifact can be shown in different windows, as done in AISA (Stein et. al., 1997). But the potential for information overload still exists.

A well-known solution to the clutter problem is to use narrow-filtering (Cavalier et. al., 1990; Malone et. al., 1995 and 1997). With narrow-filtering, a viewer looking at an artifact sees only a subset of the annotations. That subset may be determined in a number of ways, such as by author or by date. A common example of narrow-filtering in software inspection is checklist-based filtering. In checklist-based filtering, viewers of a document may choose to filter annotations by checklist entry, thus reducing the clutter problem.

In summary, clutter is a well-known problem in information theory, and the concept of narrow-filtering to reduce clutter has been shown to be useful in many circumstances. We propose that narrow-filtering is a good method to reduce clutter in annotated artifacts as well.

3.3 Delocalization

We described Soloway's (1988) classic delocalization problem in Chapter 1, and suggested how it applied to a general class of annotated artifacts. In this thesis, we define the general delocalization problem as follows:

Delocalization occurs when Items within the set of artifacts composing a system are found in multiple locations within some artifact or group of artifacts.

Thus defined, delocalization is a problem that transcends source code to include other software artifacts. For instance, in field studies of software inspection of designs, we have observed that when people recognize an error in the interface between two Items in a system, and those Items are on different Views, some people mark the error by attaching an annotation to one Item, and other people mark the same error by attaching an equivalent annotation to the other Item (Stein, et. al., 1997).

Delocalization (although not delocalization of annotations) is one component of the "Year 2000" (Y2K) problem (Jones, 1998; Yourden et. al., 1998). Conceptually, the Y2K problem occurs when in the artifacture of a system (in this case, the set of all programs that must work together), a date Item fails to represent dates from January 1, 2000 onward correctly, even though it represents earlier dates correctly. Delocalization enters into this because the same date Item may appear in many different artifacts, in many different formats. To successfully solve an instance of the Y2K problem, all occurrences of this date Item must be found and corrected as necessary.

The delocalization problem is especially severe for viewers of OO software (MacDonald et. al., 1996). When inspecting or browsing OO software, a viewer is effectively trying to determine the behavior of a software system with substantial dynamic binding while looking at static source code. This property makes it relatively difficult to effectively inspect OO software, especially when dealing with parameterization of classes and with polymorphism. Polymorphism is a special problem, because it is not clear from the static code structure what function will actually run. Thus, a viewer of an OO class (in any development phase) may be interested in seeing annotations on the base or derived classes of the class being viewed.

Delocalization even transcends documents and computer systems. Consider a replacement part for an automobile, say a windshield wiper blade. Define the set of artifacts of interest to be the physical automobile, its owners manual, and the set of all mappings from manufacturers of replacement parts that map the make, model, and year of the car to the part that fits it. In order for a consumer to get the correct part on the first try, both of the following must be true: 1) The blade type specified in the owners manual must be the one actually on the car, and 2) The replacement part manufacturer’s mapping of blade type to replacement blade model number must be correct. If either of these relationships is incorrect (e.g., because of a misprint in the owners manual) the consumer will have a problem getting the correct part. The first problem is an instance of longitudinal delocalization between the car and the owner’s manual (its documentation). The second problem is an instance of lateral delocalization between a blade type and a blade number.

In this thesis, we confine our attention to delocalization problems involving annotations made on computer systems. In general, an Item *I* in a system may have relationships to other Items that are not visible to a viewer in the same View *V* as Item *I*. Some annotations on these other Items may be of interest to viewers of *I*. A viewer who sees annotations on the other Items while looking at *I* may be able to identify problems that exist in one of the other Items, or to identify unforeseen problems with *I*.

To address the delocalization problem for annotations, we propose to apply broadening filters. Such filters would allow a viewer of an Item to see not only annotations on the Item itself, but also annotations on related Items. Annotations on those related Items are what we term foreign annotations. Essentially, we allow a viewer of an Item to see a superset of the annotations associated with that Item.

We identify three different types of delocalization relationships among and within Items. We taxonomize these relationship types below. To illustrate our taxonomy, we incorporate a running example of an annotated artifact into this taxonomy. Our example follows the conventions of UML, because UML is becoming a standard method for specifying software systems. Our example contains a set of requirements specified in UML "use cases", a design specified by an OO class diagram, and an implementation in a generic OO programming language. When necessary, we differentiate between different versions of the same artifact. In particular, we differentiate the inspected version of an artifact from the post-inspection version which contains corrections made to the defects that were identified during the inspection.

3.3.1 Lateral Delocalization

Software artifacts are related *laterally* when they are at the same development phase. Examples include two customer requirements or two different source code files. The classic delocalization problem involves lateral delocalization among source code files (Soloway et. al., 1988). Lateral delocalization often involves items of different scope. We have addressed lateral delocalization previously, by proposing that viewers of an Item *x* may benefit by seeing annotations on other Items laterally related to *x* (Stein et. al., 1998).

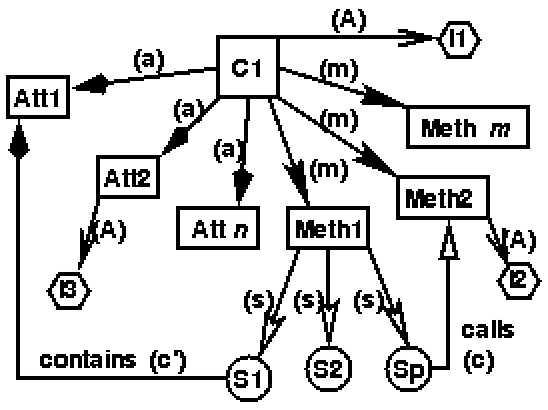


Figure 3.3: Implementation Items for class *C1*. Includes annotations I1, I2, and I3. Includes edges for method calls (c) and "contains" for use of an attribute in a statement (c').

Figure 3.3 shows the implementation of an OO class *C1* whose *n* attributes (*Att 1, …, Att n)* are related to *C1* by edges of type "attribute of" *(a)*, and whose *m* methods (*Meth 1, …, Meth n*) are related to *C1* by edges of type "method of" *(m)*. For method *Meth1* we further show its statements, related to the method by edges of type "statement of" *(s)*. The edges described above may connect Items visible within the same or different Views.

Two types of edges connecting Items in Figure 3.3 are unlikely to be visible to a viewer within a single View. These edges relate a statement of a method to another method called within that statement *(c)*, or to an attribute manipulated by that statement *(c')*. These edges indicate likely delocalization relationships. In Figure 3.3, Statement *Sp* of *Meth 1* calls *Meth 2*, so there is a lateral delocalization relationship between *Sp* and *Meth 2*. That is, to understand what statement *Sp* does, a viewer of *Meth 1* might need to understand *Meth 2*, and might find annotations on *Meth 2*, such as *I2*, to be pertinent.

Another example of lateral delocalization would occur in a high-level diagram of multiple Items at the same level of abstraction, such as a design-level class diagram. Figure 3.4 shows a set of five design classes and their interrelationships. Delocalization occurs along such relationships. For instance, a viewer of class *C3* might want to see annotations on its base class, *C1*.

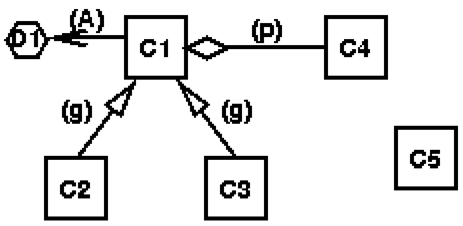


Figure 3.4: Relationships among design class Items. Includes “aggregation” (p) [part-of] and “generalization” (g) [inheritance]. Lateral delocalization occurs among related classes; annotation D1 might be pertinent to a viewer of class *C3*.

3.3.2 Longitudinal Delocalization

Longitudinal delocalization occurs between different development phases of a software system, such as between requirements and design, design and code, or requirements and a test plan. Delocalization occurs because determining the quality of a development artifact at one phase may require understanding one or more artifacts at another phase. For instance, the correctness of a design cannot be determined without knowledge of the requirements.

We propose that viewers of artifacts may benefit by seeing annotations of longitudinally related artifacts. For instance, programmers implementing a design could access the annotations on the design to try to understand design rationale. Authors of a test plan for customer acceptance testing might want to access customer annotations to the requirements to gain greater understanding of how to test those requirements.

Traceability is the ability to follow a requirement from its statement in a specification, through the design (to find which portions of the design address that requirement), and into the code (to find out where that requirement is implemented). Traceability may also refer to the linking of tests in a test plan to the requirements that those tests verify. Traceability is enhanced by viewers’ being able to see longitudinally delocalized annotations. Even in the absence of annotations, hyperlinkage of different development artifacts aids traceability by allowing viewers to use hyperlinks to follow a requirement through its realization in the design to its implementation in the code (Corriveau and Hayashi, 1994). Adding interconnected annotations to such hyperlinked documentation can further increase the usefulness of hyperlinkage by allowing viewers of one artifact to see problems that may be present in a related artifact at a different level of abstraction, without having to navigate to that related artifact first.

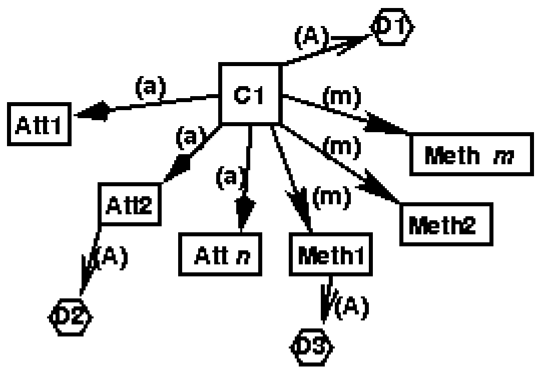


Figure 3.5: Design of Class C1. Each Item is connected to the identical implementation Item from Figure 3.3 by an "Implements" edge (not shown). The same edge types connect a class to its components in the design and implementation figures (Figures 3.5 and 3.3, respectively).

Figure 3.5 shows the design corresponding to the implementation of Figure 3.3. The relationship between these figures illustrates longitudinal delocalization. Each Item in Figure 3.5 is connected to the Item of the same name in Figure 3.3 by an edge of type "Implements" (not shown). Implementers of class *C1* might be interested in the annotations *D1, D2,* and *D3* placed on the class and its constituent parts during the design phase.

Figure 3.6 also exhibits longitudinal delocalization. The requirements of our sample system are given as a set of use cases. The high-level design is a set of classes, each of which is involved in the satisfaction of the requirements stated within some use case. These "Implements" relationship edges extend from a use case to all classes involved in implementing that use case. Designers might be interested in annotations on the use cases that their portion of the design implements. Thus, a designer of class *C3* might find annotations R1 and R3 pertinent.

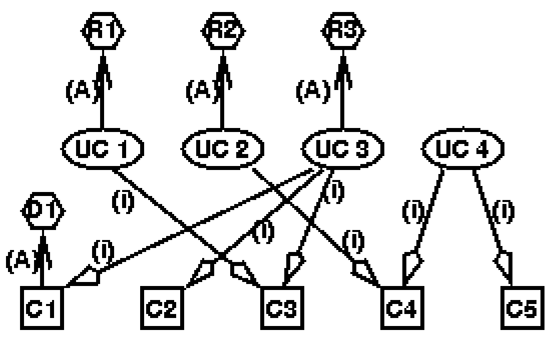


Figure 3.6: Specification/Design Mapping. Use Cases in the requirements are mapped to design classes by edges of type “Implements”. We show annotation D1 in this diagram because its scope is the entire class, and this View contains all use cases and design classes.

3.3.3 Historical Delocalization

Historical delocalization occurs between different versions of the same artifact. It is a form of organizational memory loss (Walsh and Ungson, 1991), in which information contained in annotations to early versions of an artifact is not easily available to viewers of later versions. For instance, suppose that developers of a software system chose a particular user interface in the first release of that system, and developers working on a later release wonder why that choice of user interface was made, or whether an alternative interface would be better. These later developers should be able to access pertinent discussions of the original user interface design to help them understand why the choice was made. Accessing earlier discussions might be especially useful if there had been significant turnover in the organization, and the people who made the original decision were no longer around to explain it.

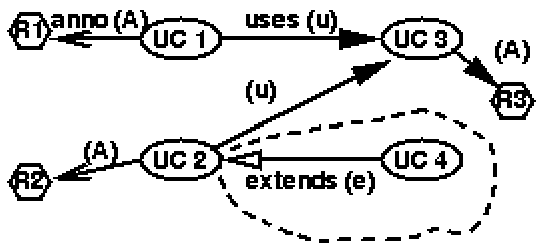


Figure 3.7: Modified Requirements. Simple requirements with edge types for “uses” (u), “extends” (e), and parameterized annotations (A). The circled portion was added after review.

Another example is given in Figure 3.7. Here the system requirements are specified at a high level by a set of use cases. The original requirements specification contained three use cases. A fourth use case was added to the requirements as a result of an inspection. To understand why this fourth use case was added, it is necessary to look at the annotations from the inspection. Someone doing software maintenance might be looking at the latest version of the requirements, but still be interested in why a given use case was added. Providing access to historically delocalized annotations can offer such a viewer the ability to understand why a certain decision was made, in the context of the latest version of the system.

Issues of new Items and deleted Items arise in historical delocalization. From one version of an artifact to the next, some Items may either disappear or be added. A common example of a new Item would be one added to provide new functionality for the system. Clearly, new Items have no history. An Item might be deleted when requirements were shed from a system. Annotations made to deleted Items will not be lost, but they may not be accessible to viewers of later versions of an artifact.

In some cases it might be possible to manually assign an annotation from a deleted Item to a different Item. For instance, it is common to "re-engineer" software systems after a number of versions have been built. This re-engineering can involve splitting or combining functions or classes (Subramanium and Byrne, 1998). Suppose that a function is split into two functions during re-engineering of a system. Annotations on that function in previous versions may reasonably be assigned to one of the functions that replace the deleted function.

Historical delocalization is characteristic of software inspection. A particular version – a "snapshot" - of the artifact is reviewed, and comments are made on that snapshot. But the released version is not the inspected version; rather it is the inspected version with defects fixed. It may be useful for future developers or maintainers to see why certain decisions were made, and so to have access to the annotations made on previous versions of the artifact. Broadening annotations to include comments made on previous versions can address these problems.

3.3.4 Research Hypothesis

Two fundamental ideas of this thesis are that delocalization is a problem for annotated artifacts, and that the delocalization problem can be addressed effectively by broad-filtering.[2] We summarize our ideas in the following research hypothesis:

*Hypothesis 1: **Broad-filtering can be used to successfully ameliorate delocalization.***

To use broad-filtering to address delocalization, we have to both build a tool that can do broad-filtering, and ascertain that broad-filtering an annotated artifact can be done in a consistent manner. Thus this hypothesis has two sub-hypotheses:

*Hypothesis 1.1: **Broad-filtering can be described mathematically in such a way that it can be applied clearly and unambiguously.***

To correctly apply a concept such as broad-filtering to a wide variety of situations, we must understand it. One way to understand a concept like this is to create a mathematical model of an annotated system incorporating broad-filtering. We hypothesize that we can build such a model, and demonstrate that it is consistent and sufficiently rich to be interesting.

*Hypothesis 1.2: **Broad-filtering can be implemented in an annotation tool.***

Above we have described the idea of broad-filtering. For the idea to be useful, it has to be implemented in some way. Thus we hypothesize that broad-filtering can be implemented in an annotation tool.

[2] We have no hypothesis concerning clutter, because the clutter problem is very well understood.

3.4 Simultaneously Addressing Clutter and Delocalization

3.4.1 Supporting Simultaneous Narrow- and Broad-Filtering

Broad-filtering of annotations, the proposed solution to delocalization, could make clutter worse by showing an overwhelmed viewer even more annotations. Thus any solution to the delocalization problem for annotations must address the clutter problem at the same time.

We propose one such simultaneous solution to the clutter and delocalization problems. Our solution is to allow viewers interested in a certain Item I to see a subset of annotations on a set of Items including both I and other items related to I. Essentially, we want users to be able to see a subset of a superset of the annotations on the Items related to I.

Viewers must be able to interleave the subset and superset properties to allow them to see the exact annotations that are useful. For instance, viewers might wish to see annotations on a virtual function or any function polymorphically related to that function, but only annotations of a specific type, such as a comment on a specific variable used within a function. Determining exactly which annotations are useful in a certain type of system is a domain-specific problem. We discuss this problem for some domains of software artifacts in Chapter 5.

To do the above interleaving, a viewer must be able to define, compose, and apply filters that transform the initial View so that the viewer sees exactly that set of desired annotations. These annotations may include some annotations that are foreign annotations in the initial View.

3.4.2 Research Hypothesis

The idea that the clutter and delocalization problems can be effectively addressed together leads to the following research hypothesis, containing two sub-hypotheses:

Hypothesis 2: *Narrow-filtering can be successfully combined with broad-filtering to simultaneously reduce clutter and ameliorate delocalization.*

Hypothesis 2.1: *Simultaneous narrow-filtering and broad-filtering can be described mathematically in such a way that they can be applied concurrently without ambiguity.*

We wish to understand the simultaneous application of narrow-filtering and broad-filtering. Thus we seek to create a rich mathematical model of these simultaneous filters that is consistent.

Hypothesis 2.2: *Simultaneous narrow-and-broad filtering can be implemented in an annotation tool.*

To be effective, we have argued above that broad-filtering to address delocalization problems must be accompanied by narrow-filtering for clutter reduction. We hypothesize that a tool supporting broad-filtering can be made to support narrow-filtering as well.

3.5 Summary

This chapter discusses the problems of clutter and delocalization. Our goal in this thesis is to address these problems simultaneously. We discuss how clutter and delocalization problems arise, and we taxonomize delocalization problems along three axes. We discuss our focus on delocalized annotations, and propose a high-level means of addressing clutter and delocalization. Finally, we state a research hypothesis related to delocalization, and a second hypothesis relating to the simultaneous interaction of delocalization with clutter.

CHAPTER 4: ANNOTATION AND SOFTWARE INSPECTION

Annotation of software artifacts is a core activity within formal software inspection. Software inspection was the original motivating application for this thesis on annotation. Having participated in over 100 software inspections, I have personally witnessed many delocalization problems, of all three varieties that we have identified in our taxonomy: lateral, longitudinal, and historical. Software inspection is also a venue in which it is possible to explore theories about the annotation of software artifacts within a formalized domain. Formal inspections are short-term phenomena that consist of a well-defined set of activities. Thus inspection is an ideal situation in which to study annotation issues.

We view software inspection as a three-part process:

1. Annotate the artifact, with each annotation representing a potential fault.
2. For each annotation, determine whether that annotation represents an actual fault.
3. Resolve the fault. Some protocols for software inspection consider fault resolution to be part of inspection (Humphrey, 1989), while others do not (Fagan, 1976).

It is clear from this annotation-centric view of software inspection that annotation plays a key role in inspection. This section discusses the specialization of filtered annotation to inspection.

4.1 Annotation and Choice of Inspection Protocol

Choices concerning the basic format of software inspection are in some measure affected by support for filtered annotations. The choices we discuss in this chapter are:

1. How should an inspection be conducted in space and time?

2. Should the Moderator attempt to "Correlate faults" before the inspection meeting to remove duplicates and merge identical faults?

3. Should an inspection address only fault collection, or should it also address fault resolution?

   3.1. If an inspection addresses fault resolution, how should it handle decision support?

4.1.1 Conducting Inspection in Space and Time

There are three possible choices for conducting software inspections in time and space. We described these in detail in Section 2.5. They are FtF inspection; distributed, synchronous inspection; and distributed, asynchronous inspection. We feel that distributed, asynchronous inspection is the preferred format for the purposes of addressing delocalization issues.

To address delocalized annotations we want the annotations to be recorded so that they can be associated with specific Items of the artifact. In all cases, annotations can be recorded with the proper tools. Thus, clutter and delocalization problems can be addressed regardless of the space-time conduct of the inspection. However, in the synchronous case, and especially during an FtF inspection meeting, people may merely discuss their annotations and not commit them to writing. Such annotations can be easily forgotten. Having the annotations stored permits the viewing of past annotations to address the problem of historical delocalization. So permitting filtered annotations strengthens the case for performing distributed, asynchronous inspections.

Previous work described in Section 2.5 suggests that aside from issues of delocalization, distributed, asynchronous inspection is similar in effectiveness to synchronous forms of inspection. So our recommendation in favor of distributed, asynchronous inspection does not exact a penalty in terms of inspection effectiveness.

4.1.2 Fault Correlation in Inspection

All formal inspection models with which we are familiar involve pre-meeting preparation of the reviewers to identify faults in the artifact being inspected. Additionally, Humphrey suggests a "Fault Correlation" phase after this pre-meeting phase but before the meeting. To do this, all reviewers formally write out their faults, and the author of the material under inspection merges their fault lists into one combined fault list for discussion at the meeting.

The perceived advantage to this phase is that it allows the author to perform two activities that may streamline the meeting. The author may remove from consideration those annotations that, in the opinion of the author, do not represent faults. The author may also merge duplicate faults that more than one person has found. A special case of this is to merge faults that were originally placed in different places, for instance on different files in the artifact. An example of merging faults occurs when inspectors find a fault with the interface between two Items A and B, and some inspectors attach the fault to Item A while others attach the fault to Item B.

We propose to not perform fault correlation when working with filtered annotations, for two reasons. First, it is dangerous to allow the author to arbitrarily remove annotations that may be critical of his/her work. Miller has recognized this, and proposed that a non-author perform fault correlation (Murphy and Miller, 1997). We go one step further and question whether it is wise to allow *anyone* to remove annotations without letting the full inspection team see them. Moreover, supporting broadened annotations reduces the need for merging duplicate faults. If people attach the same fault to different Items in an artifact, there is likely to be some relationship between those two Items. If the delocalization relationships are properly defined, all relevant annotations are visible and can be addressed from either Item. That is, if there is an interface between two Items A and B, then annotations on A should be broadened to be visible from B, and conversely. So it ceases to matter whether the fault was placed on A or on B.

Fault correlation may still have value in merging duplicate faults placed on the same Item, but overall the use of filtered annotation reduces the need for fault correlation.

4.1.3 Fault Resolution as a Phase of Inspection

Fagan’s traditional philosophy is that the inspection meeting should concern itself with fault collection, and should have as an output the list of faults (Fagan, 1976). Resolution of those faults is beyond the scope of Fagan inspection. Gilb and Graham (1993) also follow this philosophy.

Humphrey’s (1989) alternative philosophy is that fault resolution is within the scope of inspection. In Humphrey’s protocol reviewers collect faults on their own and submit a fault list to the author of the artifact before the inspection meeting. The purpose of the meeting in this protocol is not only to agree on whether the proposed faults are actual faults, but also to resolve as many faults as possible.

Our research group has traditionally used the Humphrey model of inspection because “it is more structured (than Fagan’s model) and provides intermediate results through the individual and correlated fault lists” (Mashayekhi, 1995).

The presence of broad-filtering to address historical delocalization adds another reason to include fault resolution as part of inspection. Annotations of software artifacts made during inspections can become part of the historical record of an artifact. The problems found during an inspection become part of the historical record for all models of inspection. If fault resolution is included as part of inspection, the solutions to the problems found also become part of the historical record.

With broad-filtering, it is possible for viewers to access historically delocalized annotations from inspections of previous versions of an artifact. They can then use these annotations to understand the rationale behind decisions that were made in the past, thus reducing the organizational memory loss alluded to in the definition of historical delocalization. So fault resolution is part of our preferred inspection protocol when using filtered annotations.

4.1.4 Decision Support for Fault Resolution

Our recommended inspection protocol with broadened annotations includes a fault resolution phase. This leads us to ask what sort of decision support should be provided to enable faults to be resolved.

AISA used a voting protocol, in which a fault was resolved by the application of a two-step procedure (Mashayekhi, 1995; Stein et. al., 1997). First, someone submitted a formal proposal for a fault resolution. Then all inspection participants except the author of the inspection material voted on the proposal. AISA could be set to accept the proposal if either a majority of the people voted for it, or if it were unanimously approved by those who did not formally abstain from voting.

In the AISA field trial, people loathed voting for fault resolution (Stein et. al., 1997). The main problem is that not everybody cared about every fault. Consider an inspection in which each inspector has a well-defined role. Some inspection models require this (Knight and Myers, 1993), but even when not required it may occur naturally. For instance, suppose that in some organization, every document sent to a customer is reviewed by a member of the legal department. Such a domain expert may not care about most faults outside of a specific domain, only about those faults within the domain – here, those with legal ramifications. This could be a small percentage of the faults. It is reasonable that domain experts would vote to abstain from the decisions on resolutions of faults irrelevant to their domains. Yet in a large document merely abstaining from many votes could be a very time-consuming and annoying procedure.

Additionally, according to the AISA model a formal proposal must be made after all the "brainstorming" is done. So someone will have to write up an extra annotation summarizing what was already said, so that it could be the subject of a vote. This is also extra work.

Thus we propose doing away with formal voting. We propose that the Moderator of an inspection close an issue when she feels it is resolved, with the resolution evident from the discussion thread. The Moderator may, if she wishes, add another annotation to summarize the resolution.

4.2 Inspection Protocol

This section discusses the inspection protocol that we use during inspections to test our hypotheses about the usefulness of filtered annotations during software inspection. Our main contribution in this section is how the participant roles and protocol states impact or are impacted by the choice to use filtered annotations. We propose this as a good protocol to use for distributed, asynchronous software inspection with filtered annotations. It is essentially the Humphrey inspection protocol (Humphrey, 1989) as modified for use with AISA (Mashayekhi, 1995; Stein et. al., 1997), without fault correlation or voting.

We classify each inspection participant into one of three roles.

Producer: An author of the artifact being inspected. An artifact may have multiple producers.

Moderator: The person in charge of the inspection. A Moderator is also a Reviewer.

Reviewer: One who inspects the artifacts for errors. An artifact may have multiple Reviewers (indeed, it is generally suggested that there should be at least two reviewers in an inspection (e.g., (Fagan, 1976)), including the Moderator).

The protocol consists of three states, which are followed in order:

Initialization: The Moderator parses the artifacts into the inspection system, generating annotatable Items.

Fault Collection: The Reviewers annotate the Items of the artifacts with potential faults.

Fault Resolution (also called *Discussion*): All participants discuss the annotations and, if possible, asynchronously resolve the faults they identified in Phase 2.

For formal inspection, a filtered annotation system must incorporate two notions. The first notion is that of *state*, since allowable actions differ with the inspection state. The second is that of participant *role*, so that an annotation may be mapped to a role as well as to an author.

Permission to create or change annotations varies with the state of the inspection and the role of the inspection participant. Table 4.1 shows what actions are permissible in what states by people in what roles.

Table 4.1: Activities during the states of software inspection.

	Annotate Items?	Change Annotation Scope?	Annotate Annotations?
Initialization	No	No	No
Fault Collection	Yes	Yes	No
Fault Resolution	No	No	Yes

During Fault Resolution, all inspection participants discuss faults to determine their resolution. These discussion comments are attached to the fault annotations, which are themselves attached to Items of the artifacts. This process can be easily described in our model. When an annotation is made it is treated as an Item in its own right. It can then be annotated itself, with these annotations becoming other Items.

4.2.1 Roles of Inspection Participants

Using filtered annotations can have some impact on the nature of the Producer, Moderator, and Reviewer roles in a software inspection. We describe these impacts by role. Table 4.2 indicates which people may perform which activities.

Table 4.2: Activities of roles during software inspection.

	Find Faults	Discuss Faults	Resolve Faults
Moderator	Yes	Yes	Yes
Producer	No	Yes	No
Reviewer	Yes	Yes	No

4.2.1.1 Producer

A Producer is not permitted to identify faults during Fault Collection, but may view all annotations. During Discussion, a Producer may freely discuss the annotations. Since there is no voting, a Producer is equivalent to a Reviewer during Fault Resolution.

4.2.1.2 Moderator

The Moderator has all the responsibilities of a Reviewer (see below), as well as special responsibilities.

The Moderator must initialize the inspection. This involves making sure that the artifact to be inspected is parsed into the system correctly. This includes selecting the type of artifact from among the parsable choices (e.g., Java source code, UML design class diagram) and assigning participants to the inspection and roles to the participants.

The Moderator also resolves faults in this protocol. That is, a fault is resolved if and only if the Moderator decides it is resolved and chooses to mark the fault as resolved. This is a major difference from the voting protocol of AISA and some other tools.

4.2.1.3 Reviewer

During Fault Collection, a Reviewer views the artifacts and annotates Items, to seek clarification and to identify faults. During Discussion, a Reviewer participates in discussion threads on the collected faults, and may propose fault resolutions. In our protocol, the Reviewers do not formally vote on the resolution of a fault; instead, the Moderator determines when a fault has been resolved, or when it cannot be resolved within the confines of the inspection. A Reviewer may also add new annotations to Items during the Discussion phase, including annotations that identify additional faults.

In some inspection protocols, such as InspeQ (Knight and Myers, 1993), Reviewers can be classified into different types, such as standards reviewer, performance reviewer, etc. Our protocol does not formally support this classification, although it does not preclude it as an enhancement. Similarly, our protocol permits the use of checklists to classify errors, but does not require it. For instance, if Reviewers use checklists to classify specific types of errors, a viewer may choose to filter annotations by the checklist error that the annotation addresses.

As another example of fine-grained role definition, domain experts may be Reviewers of many documents, but only be interested in a small part of each one. They need only participate in resolutions of those faults that affect their domain. By giving the Moderator authority to decide when a fault is resolved we eliminate voting, thus making it easier for domain experts to concentrate on a small part of an inspection.

A major motivation for this thesis was the to address delocalization that arises among Reviewers during Fault Collection. A common occurrence in this phase is for different Reviewers to make the same annotation that identifies a fault, yet make this annotation in different places (on different Items). By broad-filtering these annotations, a fault found by a REviewer viewing one Item can be made visible to Reviewers viewing other, related Items.

For instance, suppose the graph model underlying an inspection has edges between a function definition and all calls to that function. By broad-filtering annotations on a function call, a fault found in one call to a function could be made visible to Reviewers viewing any other call to that function, and to Reviewers viewing the function itself.

As another example, by viewing the design for source code, and broad-filtering annotations to include foreign annotations on the design, a Reviewer might determine that the code being reviewed fails to implement the design.

Additionally, in a large system, every software module has an owner responsible for that module's correct functioning. With filtered annotations, the module owner could see faults found in other modules that might potentially interact with their module.

4.2.2 Research Hypothesis

Since software inspection is a major, well-understood application of annotated software artifacts, we believe that the ability to utilize filtered annotation within software inspection is an important proof-of-concept for the idea of filtered annotation. This belief motivates our third and final research hypothesis.

Hypothesis 3: *People can make use of an annotation tool containing narrow-filtering and broad-filtering for software inspection.*

To demonstrate the usefulness of simultaneously broad- and narrow-filtered annotations, we hypothesize that computer scientists, students, and/or software developers who are given a tool that supports filtering of annotations will use those filters to make their annotation job easier. We also have two sub-hypotheses concerning ways in which this would work.

Hypothesis 3.1: *Reviewers in a software inspection will find faults during the Discussion by seeing foreign annotations.*

Consider a Reviewer who sees a foreign annotation A while looking at some Item I that is related to, but different from, the Item that A annotates. We hypothesize that seeing such an annotation may cause the Reviewer to think differently about I, and possibly to recognize a heretofore undiscovered fault in I.

Hypothesis 3.2: *Reviewers in a software inspection will add to the discussion threads of foreign annotations.*

People will be able to respond to foreign annotations as effectively as they respond to native annotations by annotating those foreign annotations, even though they are not viewing the annotated Item at the time of response.

The major hypothesis concerns both narrow-filtering and broad-filtering, while the sub-hypotheses superficially concern only broad-filtering. However, narrow-filtering indirectly enters into these sub-hypotheses because without narrow-filtering, viewers would be overwhelmed with annotations, and might be unable to effectively use the foreign annotations, or find the relevant discussion thread to which they want to contribute.

4.3 Filtered Annotation and Johnson's Future of Inspection

As mentioned in Chapter 2, our work addresses four of Johnson's requirements for enhanced formal inspection (Johnson, 1998).

"Build organizational knowledge bases on review." One impediment to building such bases is historical delocalization, since annotations made during inspection are associated with the inspected version of artifacts. By addressing historical delocalization, we allow annotations made during inspection to be part of the knowledge base.

"Break the boundaries on review group size." Narrow-filtering to reduce clutter can permit review groups to be large without overwhelming participants with irrelevant annotations.

"Shift the focus from defect removal to improved developer quality." One way to do this, Johnson writes, is to bring inspection beyond raising issues to resolving them, since "focused issue resolution discussions are a high quality, efficient, and effective means for learning." Our annotation model (and the system we are building to support it) includes a threaded discussion subsystem for resolving faults found. Moreover, our solution to the historical delocalization problem makes such discussions accessible in the context of the latest version of the software product. This gives developers access to the knowledge base of previous reviews in the context of the actual artifacts they are working with.

"Outsource review and insource review knowledge." Johnson discusses using external organizations (such as an internet-based user interface review service) as expert consultants within reviews. This approach introduces security concerns, since all inspection participants can normally see each others' comments.

We propose that narrow-filtering can be used as a form of access control. At the time an inspection is initialized, the Moderator could define a filter for certain users that would only permit them to see the annotations from a subset of the review participants. This is a straightforward application of narrow-filtering. Such a filter could be automatically composed with other, customer-defined filters when determining which annotations to show.

A similar application of narrow-filtering can be used in artifacts for which customer feedback is desired. Consider an artifact containing requirements for a new product. The developers of this product might want feedback from many potential customers, without these customers being aware of each other's annotations or even each other's existence. In our framework, an annotation filter could be defined at inspection initialization time that permitted a viewer in the Customer role to see no annotations by any other viewer in a Customer role.

4.4 Summary

Software inspection is an annotation-intensive activity. Problems we have observed in performing and studying software inspection have allowed us to recognize clutter and delocalization problems. We discuss how the use of filtering to reduce clutter and ameliorate delocalization influences the protocol for software inspection, and also how to effectively conduct an inspection when using a tool that would simultaneously address the clutter and delocalization problems as discussed in Chapter 3. We summarize our ideas in a research hypothesis on the effectiveness of filtered annotation in software inspection.

CHAPTER 5: MODEL OF ANNOTATION

Previous chapters have discussed the problems of clutter and delocalization, and have focused on the appearance of these problems in annotated software systems. We have proposed narrow-filtering and broad-filtering to alleviate these problems, and have hypothesized that these forms of filtering are widely applicable.

One means of demonstrating such wide applicability is to mathematically define software systems and their annotations, and filters, as part of a general graph-theoretic model of annotation that incorporates filtering to alleviate clutter and delocalization problems. By doing this, we demonstrate that our proposed filtering scheme is theoretically viable.

In this chapter we define a graph whose nodes are the Items we want to annotate and the annotations on those Items, and whose edges connect the nodes in various ways. Then we define filters to manipulate the edges of the graph, altering the connections among the nodes to affect the annotations that a viewer sees when looking at an artifact. We also state and prove some theorems that suggest these manipulations can be performed in practice and are useful.

This chapter describes the model in natural language for ease of general understanding. This model can also be specified more formally. Appendix 3 is a formalization of this model using the Z specification language (Diller, 1994).

5.1 Graph-Theoretic Model

Throughout this thesis we refer to “software artifacts”. Informally, an artifact is a human-created document that supports software activity, such as a high-level design document. The scope of a single artifact is that portion of the human-created documentation that we choose to consider as a whole at a given time for a given purpose. For instance, in some cases the source code for a system might be treated as one artifact, and in other cases each source code file might be treated as a separate artifact. Thus, the scope of the term artifact is very context-dependent. But we can talk more rigorously about the set of all artifacts for a system. We refer to this set of all artifacts as the *artifacture* of a system.

Definition 1: The *artifacture* of a system is a directed graph $G = \{ I, L \}$, where I denotes the set of nodes (or "Items") of the graph and L denotes the set of edges (or "links") of the graph connecting the nodes.

With this definition, we do not require that the graph even be connected. For instance, suppose that the artifacture for a system consists of a specification, a design, source code, a test plan, and a user manual. Each of these five artifacts might be self-contained, so that the artifacture of the system could be divided into five connected subgraphs. Or some of the artifacts might be connected. For instance, the specification might be connected to the test plan, but the code might not be connected to the user manual.

5.1.1 Items and Views

Definition 2: An *Item* is a node x of G. The set of Items can be partitioned into two disjoint classes: (1) the set of *production Items (PIs)*, those Items that are part of the non-annotated artifacture, and (2) the set of *annotations*, those Items that are nodes added by people to the PIs, or to other annotations, for the purposes of comment or explanation. If we denote the set of PIs by Π and the set of annotations by A, then the set of nodes $I = \Pi \cup A$.

In this chapter, the terms "Item" and "node" are interchangeable, as are the terms "link" and "edge". We refer to nodes and edges when we want to emphasize the graph-theoretic structure of the model, and we refer to Items and links when we want to emphasize the content of the graphs.

Definition 3: A *production subgraph* is a subgraph of G containing only PIs. The set of all PIs in G is the *production system*.

We use the distinction between a production subgraph and annotations when computing Views. The disjoint nature of PIs and annotations is vital to our ability to mathematically manipulate the model. It is a weak restriction, because it merely means that a PI cannot become an annotation, and conversely.

The alternative would be to allow such conversion. For instance, we might say that in source code, comment lines annotate other, non-commentary Items in the code. Then the commentary could be both PI and annotation. Additionally, we might say that an annotation could become part of the document. Suppose someone finds a defect in a document, and proposes a solution to that defect in an annotation. We might make the annotation part of the document. But we do not do any such conversion, because it greatly complicates our graph-theoretic model, and it is unnecessary. If an author of an artifact wishes to place the text of an annotation verbatim into the artifact, the author can create a new PI to hold that information.

Definition 4: An *atomic Item* is an Item that cannot be broken down into constituent Items. That is, x is an atomic Item of G if and only if there is no other Item $y \in G$ such that $y \subset x$. An Item that is not an atomic Item is a *composite Item*.

We choose the smallest PIs of a system that we wish to be able to annotate as our atomic Items. This choice may vary with different annotation events of the same system. Thus two separate organizations could take some publicly available documentation, and choose to define atomic Items differently. For instance, in natural-language text one group may choose a word to be an atomic Item, while another group may choose a letter to be an atomic Item, with a word becoming a composite Item made up of letters. Admittedly, this latter choice of atomic PIs is rather pathological.

We also permit there to be parts of the artifacture that are not themselves atomic Items, but are part of some composite Item. For instance, if we made words atomic PIs in natural-language text, then we might make paragraphs composite PIs. Yet paragraphs contain not only words, but also punctuation marks. These punctuation marks need not be atomic PIs (unless we want to be able to annotate them), but they are a necessary part of the paragraph.

We also consider the atomicity of annotations. An annotation on a PI is an atomic Item. Our model is cleanest if we define any reply to an annotation to also be an atomic Item, but define an annotation together with the transitive closure of its replies to be a composite Item.

The classification of Items into PIs vs. annotations is independent of the classification of Items into atomic Items vs. composite Items. As seen above, we explicitly allow there to be atomic PIs, composite PIs, atomic annotations, and composite annotations.

Definition 5: The *kernel* of a composite Item x is that subset of the transitive closure of its constituent Items such that x is visible if and only if every Item in the kernel is visible.

The definition of a kernel for a given type of Item will be system dependent. We define the kernel because we may want to annotate an Item from a View even when only part of that Item is visible, as we discussed in Chapter 3. Also, many tools allow one to elide an Item, yet annotations on an elided Item should still be visible. For instance, many design tools, such as Rational Rose (Rational Software) allow users to selectively hide parts of a composite Item, such as hiding the attributes and methods of an OO class. If visibility is to be a useful concept, such hiding must not render the Item invisible for annotation purposes.

Examples of types of Items and their kernels include the following: In a UML diagram, the kernel of a class might be the box containing the name of the class. The kernel of an annotation might be its subject, author, and date, and may possibly include the first few words of the body of the annotation. The kernel of a word of text would likely be the entire word.

Definition 6: A *View* V of G is a subset of the Items of G: $V \subseteq I$. A particular Item x is *visible* to a viewer of V if $x \in V$.

A View is a very flexible concept. Any arbitrary set of Items may form a View. In general, a View is the mathematical analogue to a portion of the artifacture that a person would normally see at once. Examples include a Web page, a chapter of a document opened in a word processing program, a single diagram from a system described with UML, or a source code file. Later we define some specific types of Views that we will use extensively.

A View is independent of its representation; the same View may be represented in many different ways. For instance, a document written in Hypertext Markup Language (HTML) may look different when viewed under different browsers, and would look different still when being edited by a word processing or text manipulation program (such as Microsoft Word or EMACS), yet these are all different representations of the same View.

If V is a View containing only PIs ($V \subseteq \Pi$), then the "traditional" annotated View V' of V is $V' = V \cup \{ y \in A \mid x \in V$ and y annotates $x\}$. V' is the View V along with its native annotations.

Definition 7: An *augmented View* $\underline{V}$ of G is a View V of G together with some links involving the Items in V.

We permit there to be more than one augmented View of a particular View V, because we can choose to augment V with any arbitrary subset of the set of all possible links among the Items of V. We will later augment our Views in some characteristic ways, and use these characteristic augmented Views extensively.

An alternative to defining a View as a set of Items would be to define a View as a set of Items together with the links between those Items. We might choose this definition because we use the links to manipulate Views. But there are three reasons that we exclude links from the definition of a View. First, a viewer can see only Items, not links. The links are implicit in any presentation of the View. Second, we gain little by adding links between the Items to the definition of Views, because when we augment Views with links, we are often interested in links between Items in the View and other Items not in the View. Since we might augment a

View in many ways, if we include links in a View we get the counter-intuitive situation where two different Views look identical to an observer no matter how she chooses to look at them, suggesting that they should be the same View. Third, defining Views by their Items makes it simple to check for equality of two Views, and containment of one View within another.

Definition 8: A *production View* is a View that contains only PIs. An *annotation View* is a View that contains only annotations. All other Views are *combination Views*.

All three types of Views in Definition 8 can be useful. A viewer will often look at a combination View of some part of an artifact together with some annotations. At other times, a Viewer might just want to see annotations, or just PIs. Finally, production Views turn out to be important intermediate results in computations that lead to showing foreign annotations.

5.1.2 Links

Definition 9: The edges L of graph G form the *links* between the Items of G.

Definition 10: Each edge in L has a set of *types*, which classify the edge along various parameters (i.e., by various properties).

One way that we will parametrize edges in this thesis is by the nature of nodes at each end of the edge. We will make frequent use of the parameterization in the following table:

Table 5.1: One useful parameterization of edge types.

Name	Symbol	Node type at tail of edge	Node type at head of edge.
"production" edge	π	production Item	production Item
"has-annotation" edge	α	any Item	annotation
"annotation-of" edge	α'	annotation	any Item
"has-foreign-annotation" edge	ω	any Item	annotation

"foreign-annotation-of" edge	ω'	annotation	any Item

Some of these types may have sub-types, in particular the production edges (of type π). These subtypes, as well as the other properties of production edges, are system dependent and are beyond the scope of this thesis. However, the types of edges incident on annotations are components of this model. Finally, an edge can have a set of types, and this parameterization defines only one type.

Since an edge may have a set of types, the type of an edge could most generally be defined to be a cross-product of this set of types. However, for annotation purposes we are only interested in this one parameterization of edges by the types of nodes to which an edge is incident. In fact, it is a fundamental axiom of our system that along this parameterization, every edge is of exactly one of the above types.

Axiom 1: Every link L of graph G is of exactly one of the types in Table 5.1 in the parameterization of edges by the types of nodes on which they are incident.

5.1.3 Paths, Distances, and Neighborhoods

The concepts of a path between two nodes, the length of such a path, and the distance between two nodes in a graph are defined in line with common usage. Within our notation, they are expressed as follows:

Definition 11: A *path* P between two nodes $x \in I$ and $y \in I$ is a set of nodes z_j that lead from x to y. Formally, $P(x, y) = \{ z_j, j = 0, \ldots, k \mid z_0 = x, z_k = y,$ and $\forall j\ (0 \leq j < k), z_j$ and z_{j+1} are connected by an edge $\lambda \in L$ }.

Definition 12: The *length* of a path P, denoted $\| P \|$, is the cardinality of that set of edges making up the path.

Definition 13: The *distance D* between the nodes of a graph is the length of the shortest path between them. Let $\square(x,y)$ denote the set of all paths between nodes $x, y \in I$, then $D(x, y) = \min \| P(x, y) \|, \forall P \in \square$.

Definition 14: The *neighborhood* $N_d(x)$ of node x is that View consisting of the set of all nodes within a distance d of node x. Thus $N_d(x) = \{ m \in I \mid D(x,m) \leq d \}$. The *neighborhood* $N_d(X)$ of a set of nodes $X \subseteq I$ is that View consisting of the set of all nodes that are within a distance d of any node in X. Thus $N_d(X) = \cup N_d(x), \forall x \in X$. We define $N_\infty(x)$ as the largest connected subgraph of G containing x, and $N_\infty(X) = \cup N_\infty(x), \forall x \in X$.

Note in particular by this definition that the neighborhood of distance zero around a node x is that node itself. Thus $N_0(x) = x$. Similarly, for a set X of nodes, $N_0(X) = X$.

Definition 15: The *augmented neighborhood* $\underline{N}_d(x)$ of node $x \in I$ is that augmented View consisting of the set of all nodes in the neighborhood $N_d(x)$, and all links among those nodes within that neighborhood. The *augmented neighborhood* $\underline{N}_d(X)$ of a set of nodes $X \subseteq I$ is that View consisting of the set of all nodes in the neighborhood $N_d(X)$, and all links among those nodes within that neighborhood.

Our purpose in defining paths, lengths, and distances was to get to the definition of the neighborhood and augmented neighborhood of a node or set of nodes. Note that every neighborhood of a set of nodes $X \subseteq I$ contains the original set of nodes X.

It becomes desirable to restrict our paths through a graph. We restrict paths by limiting them to using edges of certain types. For our purposes, an edge type can include any boolean function of the parameters of an edge. For instance, one edge type might be a "has-annotation" edge with a date parameter no earlier than June 14, 1998 and an Author parameter of Michael Stein.

Definition 16: A *restricted path* P^t between two nodes $x, y \in I$ is a set of nodes z_j that lead from z_0 to z_k along edges of type t. Formally, $P^t(x,y) = \{ z_j, j = 0, \ldots, k \mid z_0 = x, z_k = y,$ and $\forall j\ (0 \leq j < k),$ and z_j and z_{j+1} are connected by an edge of type $t\}$.

Definition 17: The *restricted distance* D^t between two nodes is the length of the shortest path between them along edges of type t. Let $\square^t(x,y)$ denote the set of all restricted paths between nodes $x, y \in I$. Then $D^t(x,y) = \min \| P^t(x,y) \|, \forall P^t \in \square^t$.

Definition 18: The *restricted neighborhood* $N^t_d(x)$ of a node $x \in I$ is that View consisting of the set of all nodes of I that are within a restricted distance d of node x, using edges of type t. Thus $N^t_d(x) = \{ m \in I \mid D^t(x,m) \leq d \}$. The *restricted neighborhood* $N^t_d(X)$ is the set of all nodes that are within a restricted distance d of any node in $X \subseteq I$, using edges of type t. Thus $N^t_d(X) = \cup N^t_d(x), \forall x \in X$.

Any restricted neighborhood of a set of nodes contains that set of nodes, since it is not necessary to traverse any edge to get to an element of that set of nodes. Thus, for all possible node types t, $N^t_0(x) = x$, and $N^t_0(X) = X$.

Definition 19: The *augmented restricted neighborhood* $\underline{N}^t_d(x)$ of a node x is that augmented View consisting of the set of all nodes of N that are within a restricted distance d of node x along edges of type t, together with those edges of type t. The *augmented restricted neighborhood* $\underline{N}^t_d(X)$ is the set of all nodes that are within a restricted distance d of any node in X along edges of type t, together with those edges of type t.

In the remainder of this section, we will use the terms "neighborhood" and "augmented neighborhood" when we are referring to both restricted and unrestricted neighborhoods. It will be clear from the notation when restrictions are in place, by the existence of superscripts.

One possible ambiguity with respect to the definition of paths, distances, and neighborhoods concerns composite Items. Suppose, for instance, that a document such as this thesis, has words as atomic Items, composed into paragraphs, composed into subsections, and so on (we need go no farther to illustrate the problem). The following figure partially diagrams the three paragraphs of this thesis beginning with "Definition 16", "Definition 17", and "Definition 18".

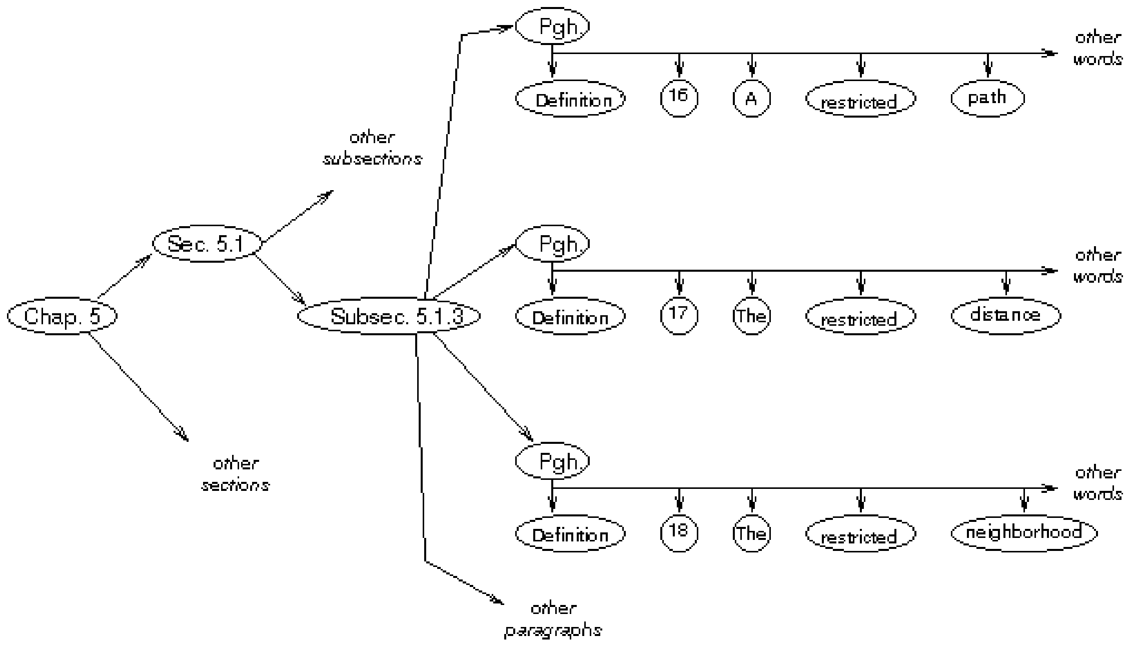


Figure 5.1: Three paragraphs of subsection 5.1.3.

Consider nodes “17” and “18”. What is the distance between them? Perhaps it is four, because one could start at “17”, trace back the edge to its paragraph, trace back the edge to the subsection, trace forward the edge to the paragraph containing “18”, and trace forward the edge to the “18”. But since a paragraph is a composite Item, and the words are part of their respective paragraphs, perhaps the distance between “17” and “18” is the same as the distance between the paragraphs containing them, namely two.

Either definition can be made to work, but we must choose one. We choose the former definition (by which the above distance is four) because it seems more intuitively sensible to compute distances at the granularity of the items at the endpoints of a path. With the latter definition, we get bizarre results when calculating distances. For instance, since an annotation is a composite Item containing the transitive closure of its replies, the distance from an annotation to a reply to one of its replies would be zero. We formalize our choice in the following Axiom.

Axiom 2: When computing distances (and thus neighborhoods) involving composite Items, we count edges that connect various Items within that composite Item.

1.1.4 Annotations

Definition 20: An annotation Item $a \in A$ is said to *annotate* (or to *be a native annotation of)* an Item $x \in I$ if there is a link from x to a of type α = "has-annotation", and an inverse link of type α' = "annotation-of" from x to a. Thus there exists a restricted path of unit length and of type α from x to a, and a restricted path of unit length and of type α' from a to x.

Definition 21: A *foreign annotation* on a PI $x \in \Pi$ is an annotation $a \in A$ such that 1) a is not a native annotation of x, and 2) there exists at least one set of PIs $e_1, \ldots, e_n$ $(n \geq 1)$ such that there is a restricted path of length greater than one from x to a, consisting of one or more edges of types $l_i \in L$ connecting PIs $x \text{ -> } e_1 \text{ -> } \ldots \text{ -> } e_n,$ followed by a single edge of type α connecting e_n to a. In this case, we create a link of type ω = "has-foreign-annotation" from x to a, and an inverse link ω' = "foreign-annotation-of" from a to x. Thus there exists a restricted path of unit length and of type ω from x to a, and a restricted path of unit length and of type ω' from a to x.

In other words, a is a foreign annotation of x if a annotates something that is related to x, but a does not annotate x itself. This definition permits an annotation a to become a foreign annotation of an Item x along multiple paths. Foreign annotations on a PI become visible through the use of broadening filters, as is detailed later in this chapter.

It is useful for people to be able to annotate annotations, thereby creating a structure akin to a discussion thread on a Usenet newsgroup. The definitions and axioms below formalize this idea.

Definition 22: Consider an annotation $a \in A$ that annotates some Item $x \in I$. If there is another annotation $r \in A$ such that r annotates a, we say that r is a *reply* to a. Let $R = \{r_1, \ldots, r_n\}, r_i \in A, i = 1, \ldots, n$ be the set of all replies to a. Then the annotation a is a composite defined by $a = b \cup R$, and $b = a - R$ is a *base annotation* of x. In general, a *discussion thread* is the transitive closure of an annotation under the "has-annotation" relationship α, whose kernel is its base annotation. A discussion thread is a composite annotation.

For our model to be consistent, we must place some restrictions on links involving annotations. We permit an annotation to be a native or foreign annotation to more than one PI, but we require a reply to be a reply to only one annotation. Also, a foreign annotation must be an annotation of a PI, not of another annotation.

Axiom 3: Let T be the set of all possible link types with respect to the parameterization of Axiom 1 (parameterized by the node types of which links are incident). Then we place the following restrictions on the links.

1. The α = "has-annotation" and ω = "has-foreign-annotation" links are the only links possible from a PI to an annotation. We further restrict the "annotates" link such that it cannot be drawn from one annotation A_1 to another annotation A_2 if there exists any link from any Item in G - A_1 to A_2.

2. The α' = "annotation-of" and ω' = "foreign-annotation-of" links are the only links possible from an annotation to a PI. Let e be a PI and a be an annotation. Then there is a link of type α' from a to p if and only if there is a link of type α from p to a. And there is a link of type ω' from a to p if and only if there is a link of type ω from p to a

Theorem 1: A discussion thread, with the α = "has-annotation" link, is a tree.

Proof: Let the base annotation form the proposed root node of the tree. Then the discussion thread is defined by edges of type α. All annotations in the discussion thread are connected, by Definition 22. To show that this is a tree, it is sufficient to show that it cannot have a cycle under relationship α. By Axiom 3, we can only create links of type α between annotations if the annotation at the head of the link has no other links, so no node on the tail of such a link can ever be part of a cycle under link α. •

We can now parameterize annotations by parameterizing the edges of type α that connect them. We will then use these parameterized edges to find the annotations meeting given criteria during our manipulations of the graph of the artifacture.

1.1.5 Filters

Definition 23: A *filter* is a function that maps one View into another View by performing a series of the following manipulations.

1. Mapping a View or augmented View into a neighborhood or augmented neighborhood of that (possibly augmented) View, to create another (possibly augmented) View.

2. Performing the standard set-theoretical operations of set union, set intersection, or set difference on two or more (possibly augmented) Views.

Thus, we define filters constructively, by describing the ways they may manipulate (possibly augmented) Views. Note that both the domain and the range of a filter is a View, but that the filter may (but need not) have augmented Views as intermediate products.

Definition 24: A *visibility filter* is a filter whose input and output Views contain exactly the same PIs, and the number of such PIs is non-zero. Thus the input and output Views differ only in the annotations they contain.

The motivation for the term visibility filter is that such a filter alters annotation visibility. Viewers will use visibility filters to manipulate the set of annotations that they can access while looking at a specific set of PIs, such as a specific document or page of a document.

Theorem 2: Visibility filters may be composed to yield other visibility filters.

Proof: (By induction.) Consider two visibility filters f_1 and f_2. f_1 maps a combination or production View into a combination or production View by definition. Let V_1 be the domain of f_1, and V_2 be the range of f_1. Then V_2 is a combination or production View because f_1 is a visibility filter, so V_2 can serve as the domain of f_2. Thus $f_2 (f_1 (V_1))$ is well-defined, and it is possible to compose two visibility filters. Now suppose that it is possible to compose *n-1* visibility filters f_1, f_2, …, f_{n-1}. Then $f_{n-1}(f_{n-2}(\ldots(f_1(V_1))\ldots))$ is a combination or production View, so we can compose it with another visibility filter f_n. •

Definition 25: An *annotation filter* is a filter whose output View is an annotation View.

Consider a production View *V* consisting of PIs that make up a "typical" View of some part of a system, say a single source code file or a page of a document. Then the difference of neighborhoods $V^* = N^{\alpha}_{1}(V) - V$ is the set of annotations of the PIs that make up *V*. Thus if *f* is a filter such that $f(V) = V^*$, then *f* is an annotation filter. The "traditional" annotated View *V'* of the system with unannotated View *V* is given by $V' = N^{\alpha}_{1}(V)$.

When creating augmented neighborhoods, we are manipulating relationships. We state the following axiom for manipulating relationships.

Axiom 4: The parameters of an edge may be manipulated independently by filters.

Axiom 4 allows us to navigate easily among the annotations. In particular, it will allow us to perform complex annotation filtering operations as a series of simpler operations.

We do not claim that the parameters of an edge are uncorrelated, just that we may treat them as though they are independent. For instance, the author and date of an annotation are unlikely to be independent, since different people annotate a document at different time periods. Or suppose one parameter of an annotation is a "type" parameter. Since different annotators are likely to have different areas of expertise, the type of annotation may correlate with the author.

The alternative to the behavior described in Axiom 4 would be to allow manipulation of one parameter to change another parameter. This occurs in a word-processing system, because changing the font style (e.g., from Times Roman to Helvetica) affects the pagination of the document. We see no need for such dependency in an annotation system, since we are interested in what is presented, not in the form of presentation; so we state that we can filter along one parameter of an edge without affecting our ability to filter along another parameter.

Theorem 3: The annotation filter that outputs the set of all native annotations of a single input Item $x \in I$ is constructable.

Proof: Begin with an Item $x \in I$. Create the augmented neighborhood consisting of all Items related to x by edges of type α, namely $\underline{N}^{\alpha}_{1}(x)$. Extract the View $N^{\alpha}_{1}(x)$. Compute the View $N^{\alpha}_{1}(x) - x$. •

Even though the above neighborhood had depth one, it still contains all discussion threads of annotations made to x because an annotation is a composite Item made up of a base annotation and all elements of the annotation tree of which that base annotation is the root node.

Corollary 3.1: The annotation filter computing all native annotations on a View is constructable.

Proof: Substitute a set of Items $X \subseteq I$ for a single PI x in the proof of Theorem 3. •

Note that Theorem 3 applies to all types of Views: essential, combination, and annotation. This allows us to generalize Theorem 2.

Theorem 4: Filters may be composed to yield other filters.

Proof: (By induction.) Consider two filters f_1, and f_2. The output of f_1 is a View created from the input Items of f_1. Let V_1 be the domain of f_1, and let V_2 be the range of f_1. Then V_2 is a View because it is a set of Items. So $V2$ can serve as the domain of f_2. Thus $f_2 (f_1 (V_1))$ is well-defined, and it is possible to compose two filters. Now suppose that it is possible to compose $n\text{-}1$ constructable filters $f_1, f_2, \ldots, f_{n-1}$. Then $f_{n-1} (f_{n-2} (\ldots (f_1 (V_1)) \ldots))$ is a View, so we can compose it with another filter f_n. •

Theorem 5: A discussion thread can be produced from a base annotation by a filter.

Proof: Let $a \in A$ be a base annotation. Annotation a is the input View for our filter. Define the augmented neighborhood of all replies to a, together with the links α between them – this is the augmented neighborhood $\underline{N}^{\alpha}_{\infty}(a)$. This is a discussion thread. Now extract the Items from the augmented neighborhood to yield $N^{\alpha}_{\infty}(a)$. •

In Chapter 3 we discussed two specific types of filters, narrowing filters and broadening filters. Now we define these within our formalism.

Definition 26: A *narrowing filter* is a visibility filter whose output View is a subset of its input View.

Since a narrowing filter is a visibility filter, the difference between the input and output Views is solely in the annotations visible. A narrowing filter thus reduces the number of annotations seen. A narrowing filter that is formed by exhibiting annotations in a restricted neighborhood of a set of PIs is thus constructable. Since an edge may be parameterized, one could construct a narrowing filter whose output includes only those annotations meeting certain criteria. For instance, such a filter might be used to only show annotations whose discussion thread contains an annotation added after a certain date.

Corollary 5.1: A narrowing filter that is formed by a composition of narrowing filters, each of which restricts the annotations in the output View along one (possibly parameterized) edge type, is constructable.

Proof: Each narrowing filter is of the form $N^{t}_{\infty}(a)$, where t is a subtype of relationship type α. Thus, by Theorem 4, their composition is constructable. •

Theorem 6: Consider a narrowing filter f that is a composition of narrowing filters $f_1 \circ f_2 \circ \ldots \circ f_n$, each f_j narrowing its input View by selecting only that subset of annotations that have certain values of some parameter p of edges of type α. The order of application of these narrowing filters is irrelevant.

Proof: Since each of the f_j is a narrowing filter, its output View contains a subset of the annotations of its input View. Annotations are Items that are on the tail of an edge of type α. By Axiom 4, the properties of an edge may be manipulated independently. Thus the order in which these annotations are filtered out is irrelevant. •

This theorem yields the common-sense result that the order of filtering annotations doesn't make a difference. So a viewer who wants to see all annotations made by John after July 21st would see the same results regardless of the order of application of the authorship and date filters.

Definition 27: A *broadening filter* is a visibility filter whose output View is a superset of its input View.

Theorem 7: Consider a PI e. Let $\hat{E}$ be the set of all PIs reachable from e by going through zero or more other PIs, thus $\hat{E} = N^{\pi}_{\infty}(e)$. Consider the set of all annotations of the Items of $\hat{E}$, given by $N^{\alpha}_{1}(\hat{E})$. Then the broadening filter f^{+} defined such that $f^{+}(e) = N^{\alpha}_{1}(\hat{E})$ is constructable.

Proof: To construct f^{+}, start with View e. Create the augmented neighborhood $\underline{N}^{\pi}_{\infty}(e)$. Extract the neighborhood $\hat{E} = N^{\pi}_{\infty}(e)$, consisting of all PIs reachable from e. Create the augmented neighborhood of all annotations on Items in $\hat{E}$, $\underline{N}^{\alpha}_{1}(\hat{E})$. •

This theorem gives us the means to identify a rich set of foreign annotations of a PI. That is, for a given PI, we can find every other PI related to it by any link between PIs that we have defined. Then we can extract all the annotations on these other PIs and attach these annotations to our initial PI as foreign annotations.

Corollary 7.1: Let $X \subseteq E$ be a set of PIs. Let S be the set of all PIs reachable from X along edges of some type Σ. Thus $S = N^{\Sigma}_{\infty}(X)$. Consider the set of all annotations to the Items of S, given by $N^{\alpha}_{1}(S)$. Then the broadening filter F^{+} defined such that $F^{+}(X) = N^{\alpha}_{1}(S)$ is constructable.

Proof: The constructability of $F^+(X)$ is immediate from Definition 25 and Theorem 7. •

The *foreign annotations* of the PIs of X are the annotations in $F^+(X)$ that are not in $N^\alpha_1(X)$, which are given by the set difference $N^\alpha_1(S) - S - N^\alpha_1(X)$. Thus the View of X with foreign annotations is $V_f(X) = X \cup (N^\alpha_1(S) - S - N^\alpha_1(X))$. To indicate the foreign annotations, one would create a link of type ω between each element of X and each foreign annotation of that element.

Corollary 7.2: The View $V_f(X) = X \cup (N^\alpha_1(S) - S - N^\alpha_1(X))$ augmented with foreign annotation links is constructable.

Proof: Consider the View of S as defined in Theorem 7, toegether with all the annotations on S, namely the augmented meighborhood $N^\alpha_1(S)$. For each PI $x \in X$, determine the set of all elements $s \in S$ that are reachable form X along edges of type Σ. This is a standard reachability problem, and can be determined by depth-first search (for instance). For every annotation a of every PI s that is reachable from x, create a has-foreign-annotation link, of type ω, from x to s; also create a "foreign-annotation-of" link, of type ω', from s to x. By doing this for all $x \in X$, we create the desired augmented neighborhood. •

By this definition, although a given annotation can only be a native annotation of one PI x, it can be a foreign annotation of multiple PIs, specifically of those PIs from which x is reachable.

Corollary 7.3: Consider the output Views of the broadening filters f^+ and F^+, augmented with the set of all their PI-to-PI links, the has-annotation link α, and the has-foreign-annotation link ω. These augmented Views contain no cycles involving the α or ω links.

Proof: Immediate from Theorem 1 and Corollary 7.1. •

Note that in Corollary 7.3, as in the rest of our formalism, we do not prohibit PIs from being involved in cycles under links between links. Rather, we merely need to restrict the links among annotations as described in section 5.1.4 so that we do not get some types of cycles involving annotations. This maximizes the breadth of PIs to which we may apply our model.

This filter F^+ is the "parent of all broadening filters". For a given input View, it contains every Item (whether a PI or an annotation) that can be reached from the input View. Thus, every filter from a given View is a subset of this filter.

Theorem 8: Consider a View V. Every View that can be constructed from V is a subset of the View constructed by $F^+(V)$.

Proof: By definition of F^+, any Item reachable along any edges of G from any node in V is part of $F^+(V)$. Therefore, if some node $x \in I$ is not in $F^+(V)$, there is no path, restricted or unrestricted, between any node of V and x, or we would have found it. Thus we cannot reach x by any of the construction methods of Definition 23. So every Item in every View constructable from V is in $F^+(V)$. Since a View is an arbitrary set of Items, this means that every View constructable from V is a subset of $F^+(V)$. •

Note that filter F^+ contains both all the native annotations and all the foreign annotations on all the Items in E. Thus Theorem 8 supports Hypothesis 1.1 of Chapter 3, which proposes that we can mathematically define a rich set of broadening filters.

With Theorems 7 (and its corollaries) and 8, we have shown that we can apply a broadening filter to determine a rich set of foreign annotations for any Item in a View, and then apply narrowing filters to these annotations (along with the native annotations) as desired. In so doing, these theorems support Hypotheses 1.1 and 2.1 of Chapter 3, which propose that we can mathematically define a rich set of interleaved broadening and narrowing filters.

1.2 Summary

In this chapter, we have described a graph-theoretic model of annotation. First we defined the basic concepts of the Items and Views of a graph, including Views augmented with the edges that connect the Items in the View. Then we discussed the links among Items, and arrived at the concepts of regular and augmented neighborhoods of Items in a View. We went on to discuss native and foreign annotations on Items, and to taxonomize the possible links between annotations and other Items (which might themselves be annotations). Finally, we discussed the concept of filters, which are the mathematical constructs by which we may manipulate the Items in a graph of a software system to extract the annotations we seek. Using broadening filters, we showed how it was possible to identify a rich set of annotations related to a given PI, thus addressing our first research hypothesis of Chapter 3. Then we showed how to combine broadening and narrowing filters to further refine the visibility of annotations for a particular PI, thus addressing our second research hypothesis of Chapter 3.

CHAPTER 6: APPLICATIONS OF OUR MODEL

This chapter illustrates the application of our graph-theoretic model to software artifacts. We show how to use our model to simultaneously reduce clutter and ameliorate the three defined types of delocalization within a software artifact. To do this, we apply our model to five examples. We stress that this chapter shows how to use our theoretical model to address clutter and delocalization in some practical examples, but we do not implement our solutions. Chapter 7 discusses how we designed and built the AnnoSpec annotation system, which performs narrow-filtering and broad-filtering of annotations.

The first illustrative example in this chapter is a UML artifact containing a system specification, a design, and some source code. This has been a motivating example for our work, and so we discuss this example in detail. This example includes many situations in which we compute the set of visible annotations using the graph-theoretic model.

The second example is a general textual document. This example is in some ways the antithesis of the UML example. Whereas the UML example shows that our model is well-suited to handling a complex but tightly constrained system, the textual document example shows that the model can be used to enhance annotation of a simple system with few defined relationships.

The remaining examples use our model to describe mathematically how some existing tools address annotations. By their nature, these tools only support narrow-filtering. By describing a range of tools we suggest that our model is powerful enough to describe annotation visibility within a diverse set of frameworks. The third example shows how inspections in our AISA tool can be handled within the context of our model. The fourth example shows how a different inspection tool, CSRS, can be described within the context of our model. Finally, the fifth example shows how a collaborative authoring tool, PREP, can be described within our model.

6.1 UML Artifact Example

Our first example is a small UML artifact that we will follow from specification through design and implementation. This example is mostly presented through diagrams. We will sometimes show two versions of the same diagram. The first will be a UML-like diagram. The second will be a set of nodes and edges in a generic graph notation. This second representation is used to illustrate the workings of our model on this artifact.

This simple artifact contains an initial requirements specification $\check{R}$. We assume $\check{R}$ becomes annotated. $\check{R}$ contains the following annotated artifacts: a use case diagram showing the use cases, and annotations of use-case scope on those use cases (Figures 6.1 and 6.2); details of each use case, diagrammed down to paragraph level, with annotations (Figures 6.3 and 6.4); and diagrams indicating filtering for clutter and delocalization, based on the previous figures (Figures 6.5 and 6.6).

The initial specification $\check{R}$ is modified to a revised specification $\check{R}$'. $\check{R}$' is not annotated. (In practice, there would likely be many more iterations, and they would continue while later phases of development proceeded, but for simplicity we assume a reviewed version of the specification and an unannotated "final" revised version.) For $\check{R}$' we exhibit a use case diagram showing the use cases (Figure 6.7), and diagrams illustrating historical delocalization (Figures 6.8 and 6.9).

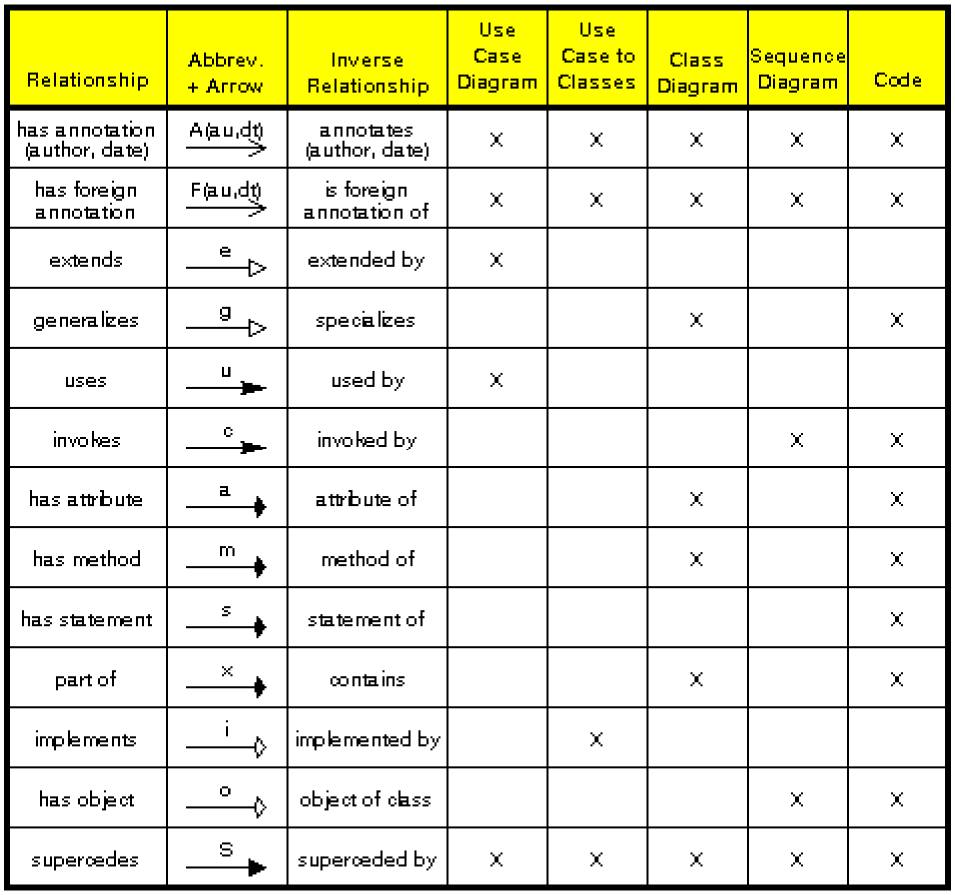
Design $\check{D}$ is the initial design based on specification $\check{R}$'. We assume that $\check{D}$ becomes annotated. We exhibit the following annotated diagrams for design $\check{D}$: the mapping from the use cases of $\check{R}$' to the design classes of $\check{D}$, showing annotations of class scope on $\check{D}$ (Figures 6.10 and 6.11); the annotated high-level design view of $\check{D}$ (Figures 6.12 and 6.13); details of one design class, annotated (Figures 6.14 and 6.15); an annotated sequence diagram for one design scenario (Figures 6.16 and 6.17); a graph illustrating longitudinal delocalization (Figure 6.18).

Design $\check{D}$ is modified to a revised design $\check{D}$', which is not annotated. For $\check{D}$' we exhibit a detail of three design classes, indicating where changes have occurred (Figures 6.19); and a graph illustrating historical delocalization between design versions (Figure 6.20).

Implementation $\tilde{I}$ is the initial implementation based on design $\check{D}$'. We assume that $\tilde{I}$ becomes annotated. We exhibit one file of annotated source code (Figure 6.21); and an annotated graph showing the relationships among the elements of the annotated source code file (Figure 6.22). We do not extend this example to a revised implementation, because that extension is not necessary to fully illustrate our ideas on filtered annotation.

Table 6.1 illustrates the relationships that we draw on the UML diagrams. On the model graphs, all arrows look like the annotation arrow, and the relationships are differentiated by letters. Also, in every case, each relationship has an inverse relationship in the opposite direction, which is not shown. Finally, note that in the AnnoSpec tool, and in any tool that we might build to address delocalized annotations, the complexity inherent within Table 6.1 would be hidden from the viewer - at least, a viewer would have the option of hiding the complexity.

Table 6.1: Key to diagrams in this section. This table contains the list of relationships used in this application of the graph-theoretic model. Some arrow symbols are reused in cases where the relationship types are similar.



Relationship	Abbrev. + Arrow	Inverse Relationship	Use Case Diagram	Use Case to Classes	Class Diagram	Sequence Diagram	Code
has annotation (author, date)	$\xrightarrow{A(au,dt)}$	annotates (author, date)	X	X	X	X	X
has foreign annotation	$\xrightarrow{F(au,dt)}$	is foreign annotation of	X	X	X	X	X
extends	$\xrightarrow{e}$	extended by	X				
generalizes	$\xrightarrow{g}$	specializes			X		X
uses	$\xrightarrow{u}$	used by	X				
invokes	$\xrightarrow{c}$	invoked by				X	X
has attribute	$\xrightarrow{a}$	attribute of			X		X
has method	$\xrightarrow{m}$	method of			X		X
has statement	$\xrightarrow{s}$	statement of					X
part of	$\xrightarrow{x}$	contains			X		X
implements	$\xrightarrow{i}$	implemented by		X			
has object	$\xrightarrow{o}$	object of class				X	X
supercedes	$\xrightarrow{S}$	superceded by	X	X	X	X	X

1.1.1 Specification $\check{R}$

Requirements specification $\check{R}$ has three use cases, two of which use the third. In the high-level use case diagram of Figure 6.1, we see that the each use case has one annotation of use-case scope. Figure 6.2 is the model's augmented View of this diagram, hereafter called simply the "augmented View", because such augmentation can only exist within the model in our framework. The caption lists the Items of the View.

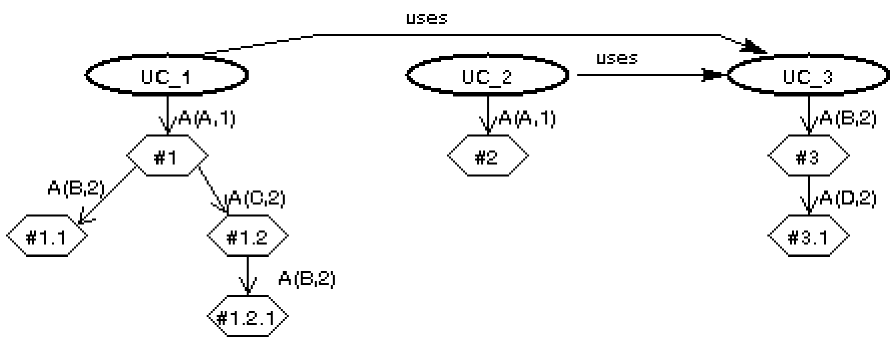


Figure 6.1: UML case diagram of the initial specification Ř. Three annotations have been made on this specification during review, two of which led to discussions.

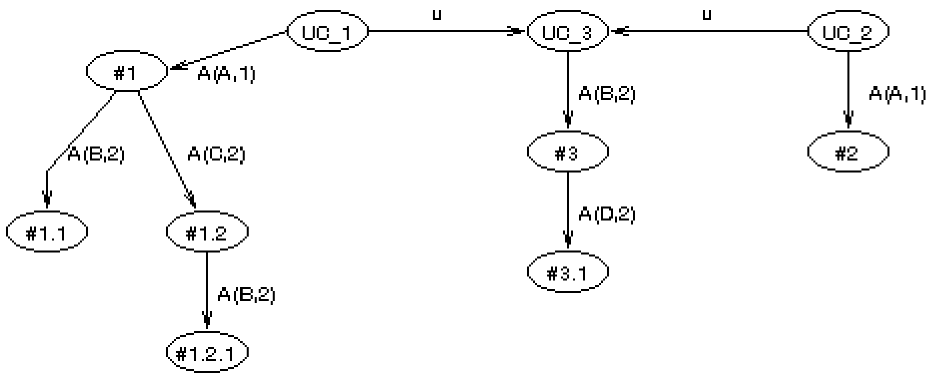


Figure 6.2: Augmented View of Figure 6.1. This diagram shows all forward relationship edges. To each forward edge of some type *t*, there is a backward edge of type *t** that is not shown. Note the parameterization of the annotation edges. Mathematically, this View of Ř is defined as $V_0(\check{R})$ = { UC_1, UC_2, UC_3, #1, #2, #3, #1.1, #1.2, #1.2.1, #3.1 }. The augmented View is merely a more traditional-looking version of the Figure 6.1. This augmented View reinforces the idea that the different types of Items and their various relationships can be treated using the same tools to compute annotation visibility.

Figure 6.3 shows the details of each use case down to the paragraph level. Figure 6.4 shows the augmented model Views, and its caption identifies the Items in the Views.

1.1.1.1 Addressing Clutter and Lateral Delocalization in Ř

Since this is the first version of the initial document of the system, there cannot be historical or longitudinal delocalization. However, there can be clutter, as well as lateral delocalization between different parts of Ř.

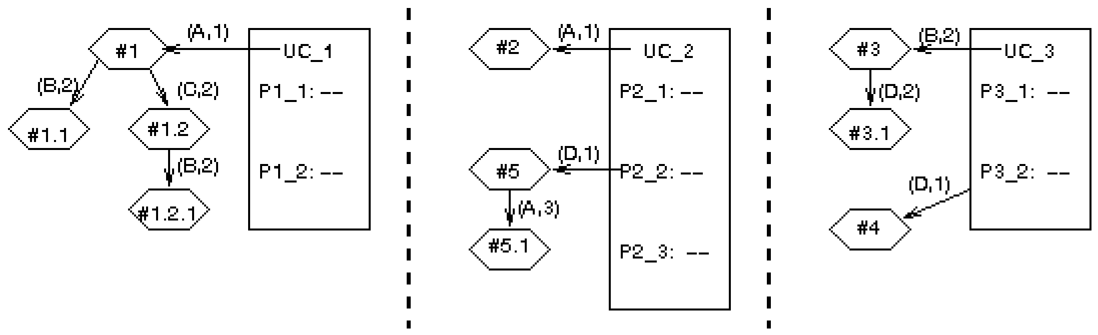


Figure 6.3: Detailed use case diagram of initial specification. This figure shows the textual description of each Use Case down to the paragraph level. Annotations made on the use cases and annotations made on the paragraphs are both visible in the unfiltered View of each use case.

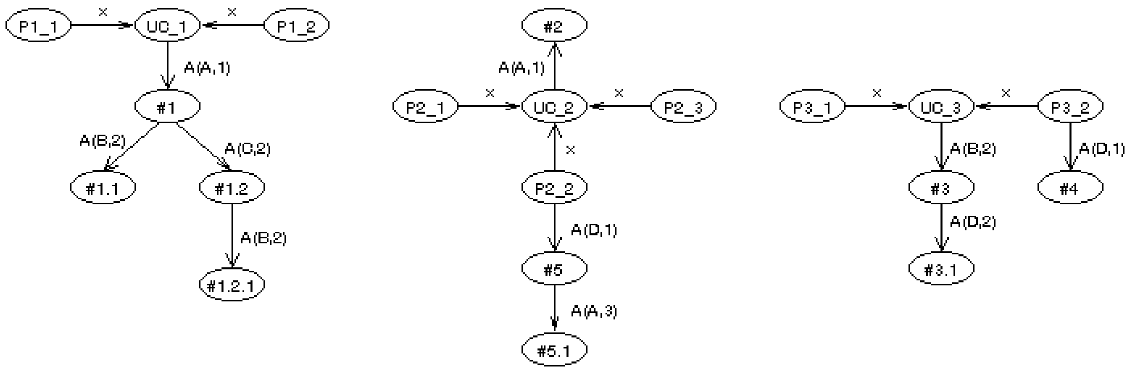


Figure 6.4: Augmented View of Figure 6.3. The Items in the View of *UC_1* are V_1 = { UC_1, P1_1, P1_2, #1, #1.1, #1.2, #1.2.1 }. The Items in the View of *UC_2* are V_2 = { UC_2, #2, #5, #5.1, P2_1, P2_2, P2_3 }. The Items in the View of *UC_3* are V_3 = { UC_3, #3, #4, P3_1, P3_2 }.

To address clutter, consider annotation (type α) relationships. Denote by $\alpha(x,y)$ the annotation relationship restricted to authors meeting criterion x and dates meeting criterion y. Use an asterisk (*) to denote no restriction for a parameter (author or date). So, for instance, annotations by Ann of any date would be denoted α(A,*), annotations by anyone after date d would be denoted $\alpha(*,\geq d)$, and annotations by Deb on date 9 would be denoted by α(D,9).

Suppose that a viewer of use case *UC_2* is interested in annotations by a particular author, say Ann. A viewer wishing to see all the annotations (if any) that Ann made on *UC_2* could start with View V_2 from Figure 6.4. V_2 is a set of nodes, so we could construct a filter to extract the neighborhood of base annotations made by Ann as $X_2 = N^{\alpha(A,*)}{}_1(V_2)$. This filter, of depth 1, returns annotation #2. Ann also made reply #5.1 to annotation #5, but this was not a base annotation – it was too deep in the discussion thread to be returned. The viewer might now be interested in further comments on those issues, perhaps including resolutions. To get these, the viewer would apply a filter to X_2, which is merely annotation #2, to obtain the entire discussion thread of this annotation. This discussion thread is given by the neighborhood $N^{\alpha}{}_{(*,*)}{}_{\infty}(X_2)$.

Now suppose that the viewer remembers that Ann said something about *UC_2*, but Ann is not available to be asked about it. The Viewer decides to look for all comments that Ann made within the discussion thread of any annotation of *UC_2*. In terms of our model, the viewer wants to see the set of all annotations made by Ann at arbitrary depth in the discussion thread of any annotation in View V_2. This set is a neighborhood of V_2 given by $Y_2 = N^{\alpha}{}_{(*,*)}{}_{\infty}(V_2)$. Y_2 contains Ann's base annotation #2, as well as Ann's reply #5.1 to annotation #5. Since replies, especially, are not useful out of context, the viewer now would want to see the full discussion threads of both annotations #2 and #5. The neighborhood for the full discussion thread of #2 was derived above. For #5, we start with Ann's reply #5.1. Now we must get the base annotation by traversing the inverse annotation relationship. Since #5.1 is a reply to a base annotation, we only need go up one level, so that the base annotation is given by $B_2 = N^{\alpha}{}_{(*,*)}{}_1(\#5.1)$. From here, we extract the discussion thread as $N^{\alpha}{}_{(*,*)}{}_{\infty}(B_2)$

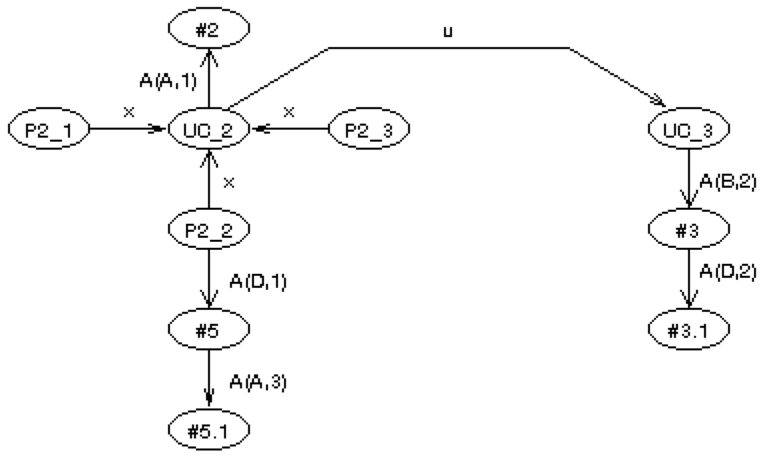


Figure 6.5: Further augmentation of Figure 6.3. This augmented View of the details of *UC_2* includes the Items found by going through the "uses" relationship to the high-level View of *UC_3* from Figure 6.2.

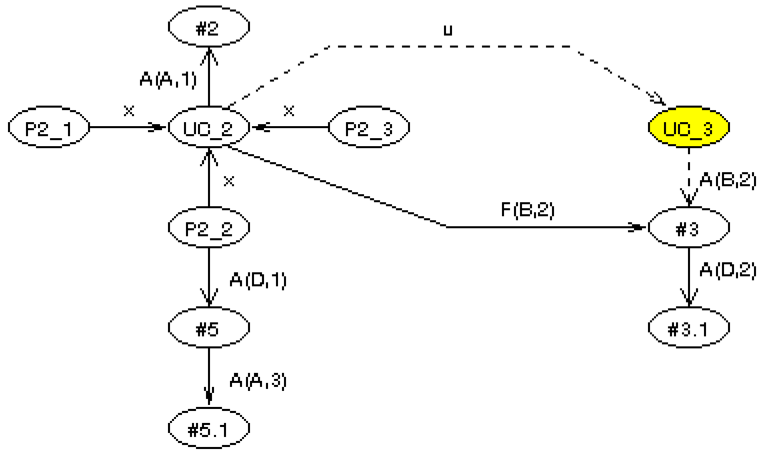


Figure 6.6: Foreign annotation of Figure 6.5. The PI *UC_3* is not part of the View *V6* augmented with foreign annotations. To choose where to attach the foreign annotation, we follow the edges in the graph of Figure 6.5.

We can also address lateral delocalization issues in specification $\check{R}$. Consider a viewer of V_2, the detailed View of use case *UC_2*. *UC_2* has a "uses" relationship with use case *UC_3*, so a viewer might be interested in annotations made to *UC_3*, because changes in *UC_3* might impact *UC_2*. To find such annotations, the viewer could first compute the filter for the neighborhood of *UC_2* that also contains PIs used by *UC_2*. This filter is $X = N^{u^*}_1(UC\_2)$, which is the set of PIs on which the viewer wishes to see annotations. To see all annotations on these PIs, the viewer computes the filter $N^{\alpha}_1(X)$. Then the viewer must replace sets of inter-PI edges in Σ followed by a single annotation edge α with a single foreign annotation edge ω from a PI in *UC_2* to a foreign annotation, following the procedure of Corollary 7.2 of Chapter 5. In Figure 6.6, we replace an edge from *UC_2* to *UC_3* followed by an annotation edge from *UC_3* to annotation #3 with a foreign annotation edge directly from *UC_2* to annotation #3. So the Viewer sees native annotation #2 and foreign annotation #3 associated with *UC_2*.

1.1.2 Specification Ř'

Annotated specification *Ř* is rewritten, incorporating the concerns addressed by its annotations. It evolves to revised specification *Ř'*, which serves in this example as a "final" specification, and is not further annotated. In this revised specification (as well as in the revised design shown later) all PIs are followed by a "prime" (e.g., *UC_2'*) to differentiate them from the eponymous PIs of the initial specification *Ř*. This is true whether or not the PIs are unchanged in the revised specification, because these PIs still belong to the latest version of the specification, even if their content has not changed with the version update.

Ř' contains a new use case, *UC_4'*, that extends *UC_3'*. *UC_3* was split into two parts: the new *UC_3'* (which contains the parts used by *UC_1'* and *UC_2'*) and *UC_4'*.

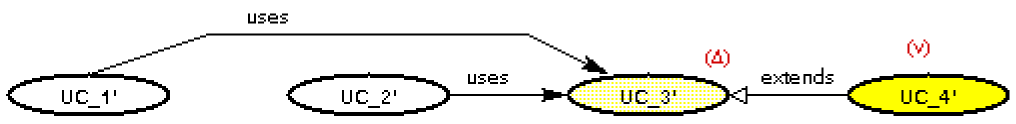


Figure 6.7: Use case diagram for the revised specification Ř'. This diagram shows how UC_3' is extended by a new use case, UC_4'. In this figure, we mark the changed use case by the (Δ) symbol, and the new use case by the (v) symbol.

1.1.1.1 Addressing Delocalization in Ř'

The use cases in the revised specification *Ř'* are related to their eponymous use cases in the initial specification *Ř* by edges of type "supercedes". This gives us access to the original use cases and their annotations. Thus we can now address historical delocalization relationships between the two versions of the specification.

For instance, suppose that a viewer of the revised specification wants to verify that the problems brought up by annotations on the original specification were properly addressed. Let View X = { *UC_1'*, *UC_2'*, *UC_3'*, *UC_4'* } be the high-level use case diagram View of $\breve{R}$'. Then to access the annotations on the original specification, one must broad-filter along the supercedes relationship to get neighborhood $Y = N^s_1(X)$ = { *UC_1'*, *UC_2'*, *UC_3'*, *UC_4'*, *UC_1*, *UC_2*, *UC_3* }, then extract the annotations on the initial specification by broad-filtering on Y to get neighborhood $Z = N^{\alpha}_1(Y)$ = { *UC_1'*, *UC_2'*, *UC_3'*, *UC_4'*, *UC_1*, *UC_2*, *UC_3*, #1, #2, #3 }. Finally, the annotations must be attached by as foreign annotations to the correct use cases in the revised specification, as shown in Figure 6.9. As before, we have obtained this result by following the method of Corollary 7.2 of Chapter 5.

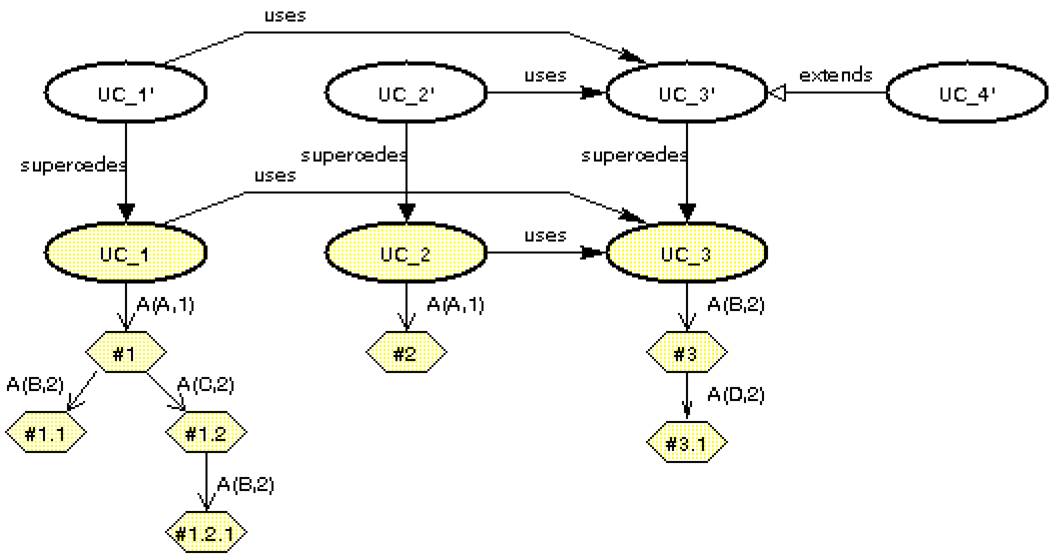


Figure 6.8: Relationship of initial and revised specifications. The shaded Items are superceded PIs from $\breve{R}$, and the annotations on those superceded PIs.

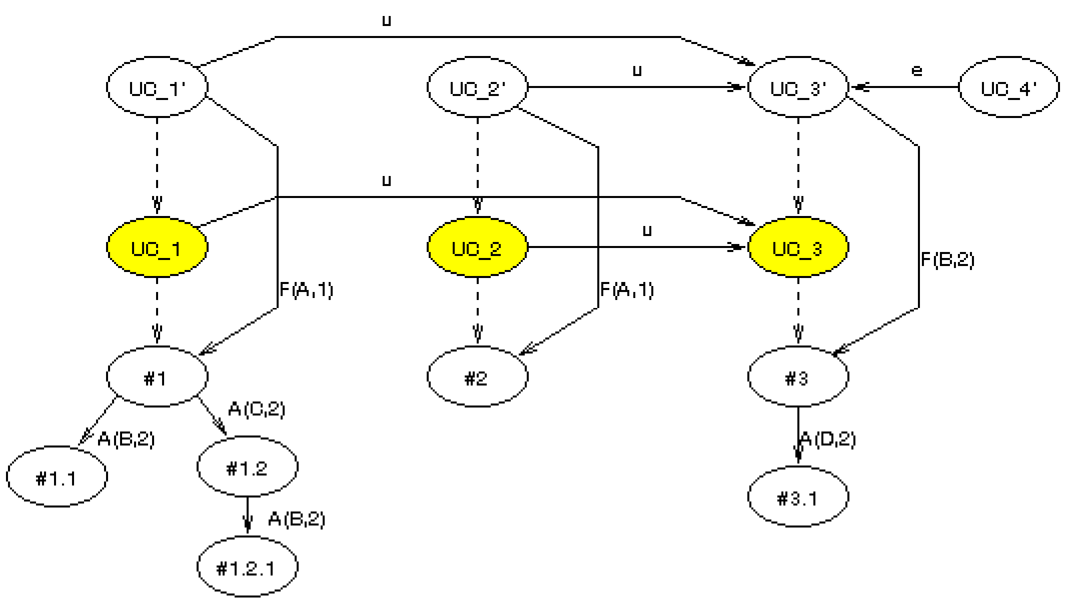


Figure 6.9: Historical delocalization in Ř. This figure shows how historically delocalized annotations from specification *Ř* are shown to a viewer of *Ř*'. We replaced some "supercedes" (S) and "annotates" (A) links in Figure 6.8 with "foreign annotation" (F) links, so that annotations on the initial specification *Ř* use cases (*UC_i)* are visible without having to view the *UC_i* themselves.

1.1.3 Design Ď

Design *Ď* is the initial design of the system, based on the specification *Ř*'. As with that specification, we describe the design graphically.

First we show how the requirements map to the design. Figure 6.10 shows the mapping of high-level design classes to the use cases they implement. As part of the design, this figure shows the annotations of class scope made on the design, although it shows no details of either the specification or the design. Figure 6.11 is the model View of this same mapping.

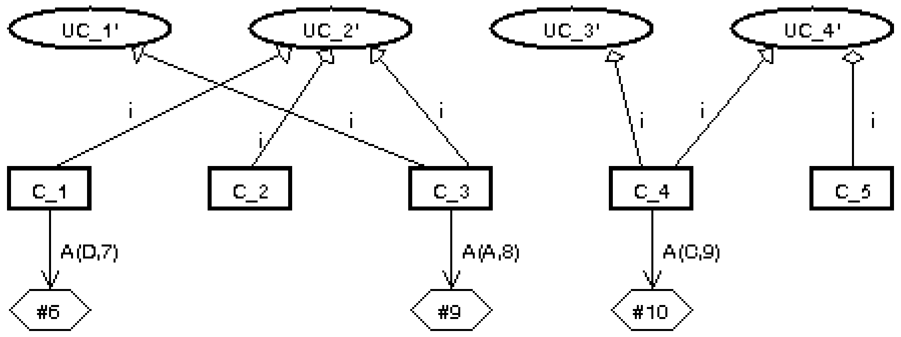


Figure 6.10: Mapping of use cases of Ř' to design classes of Ď. Annotations made on *Ď* at class scope are visible in this View.

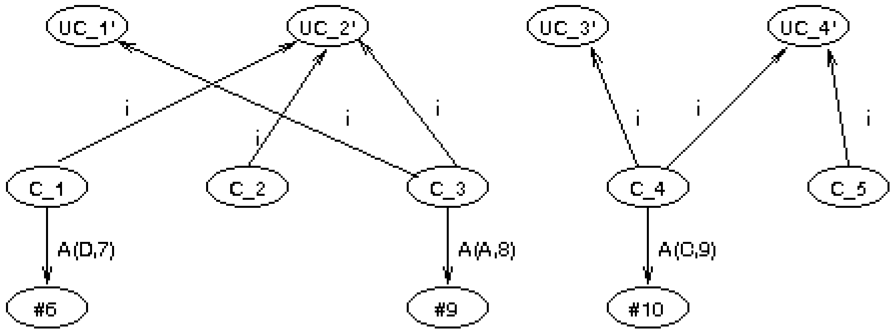


Figure 6.11: Model View of Figure 6.10. This figure shows the mapping of use cases to design classes from the graph-theoretic model perspective. This View V_{sd} = { UC_1', UC_2', UC_3', UC_4', C_1, C_2, C_3, C_4, C_5, #6, #9, #10 }.

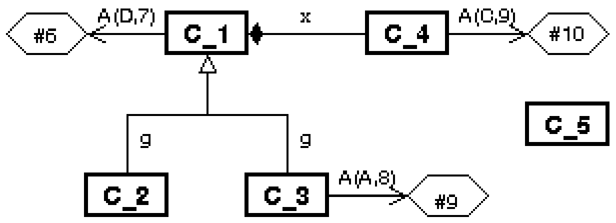


Figure 6.12: High-level design view of *Ď'*. It shows the classes but not their internal structure. Thus, it shows only annotations of class scope on the design. If we call this View V_d, then V_d = { C_1, C_2, C_3, C_4, C_5, #6, #9, #10 }.

Figure 6.12 is the high-level class diagram of design *Ď*. It shows the links between classes and the annotations of class scope. Its purpose is to show the interconnections of the classes without showing any details. Figure 6.13 is the model View of this high-level class diagram.

For our purposes in this thesis, a "detailed" design for a class is a class diagram that shows the class, its attributes, and the signatures of its methods, but does not give details of the implementations of the methods. In that spirit, Figure 6.14 is a detailed diagram of three of the design classes, and Figure 6.15 is the model View of the detailed design of those three classes.

Many other forms of diagrams may be used in the UML to specify a design, all of which we can map into our model. We produce only one such diagram here. A "sequence diagram" is a diagram that shows which objects of which classes invoke which other objects of the same or other classes. Figure 6.16 is a sequence diagram showing the sequence of method invocations for some scenario in the design. Figure 6.17 is the model View of that scenario. The sequence diagram is important because it shows how we address the "calls" relationship, which is an important relationship in many software artifacts, even if they are not object-oriented.

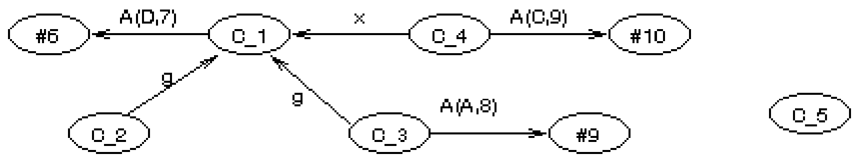


Figure 6.13: Graph-theoretic model version of Figure 6.12.

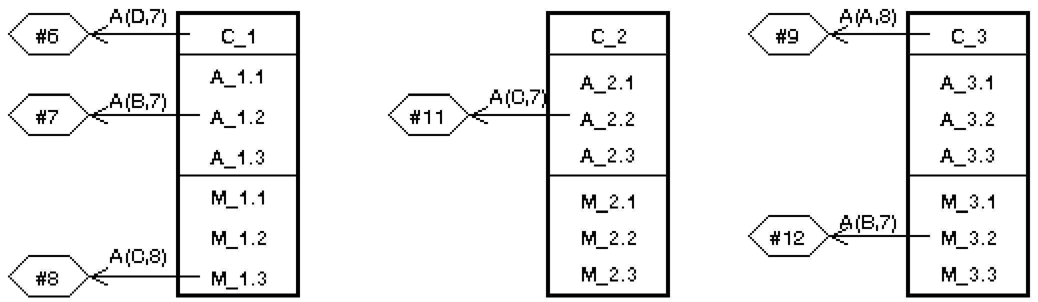


Figure 6.14: Detailed design of classes C_1, C_2, and C_3 in design *Ď*. It shows annotations of attribute and method scope. In many development tools it is easy to switch between views of a class at a high level and a class with details shown.

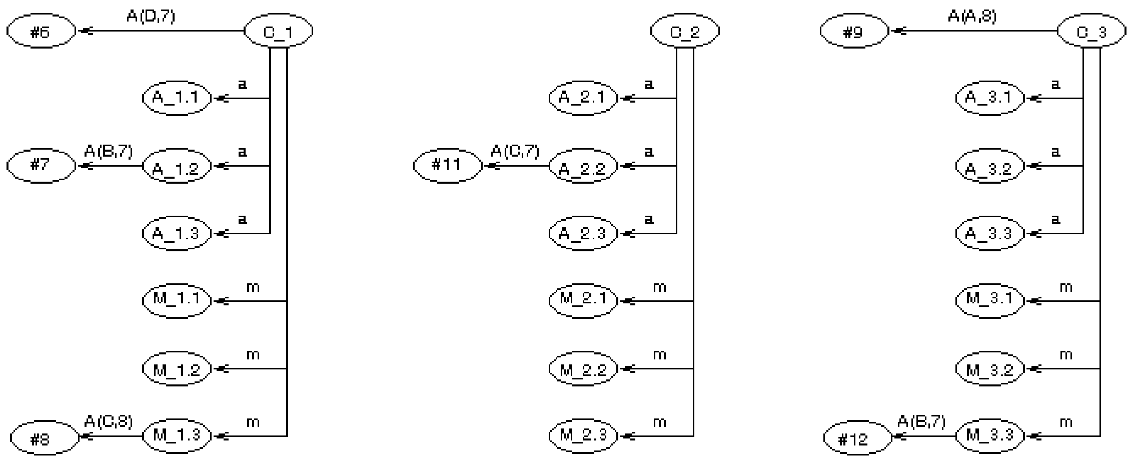


Figure 6.15: Graph-theoretic model version of Figure 6.14. View V_{C1} of C_1 is given by V_{C1} = { C_1, A_1.1, A_1.2, A_1.3, M_1.1, M_1.2, M_1.3, #6, #7, #8 }. The equivalent Views of classes C_2 and C_3 may be determined analogously.

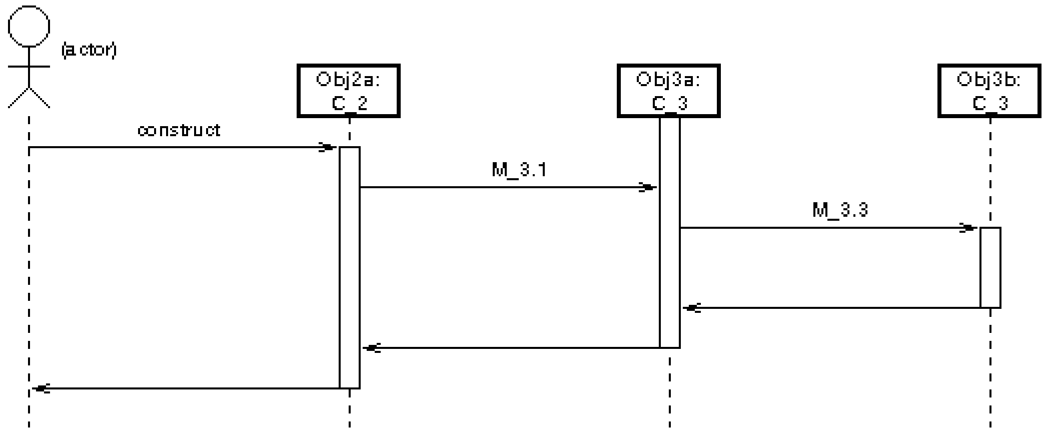


Figure 6.16: Sequence diagram for design *Ď*. It shows an "Actor" (an external agent to the system) constructing an object of class C_2 that invokes a method on an object of class C_3, which in turn invokes another method on a different object of class C_3.

1.1.1.1 Addressing delocalization in Ď

Having shown both a specification and a design, we can now address longitudinal delocalization. Consider a viewer of the specification who is interested in the design of the system that will implement certain of the use cases. Such a person might like to be aware of any annotations made on the design classes, since these annotation could indicate problems in meeting the requirements. Suppose someone looking at a high-level use case diagram, such as Figure 6.7, wishes to see all annotations of class scope on the classes implementing the use cases. These annotations can be shown as foreign annotations associated with the use cases whose classes implement those use cases, by using the following procedure. Figure 6.18 shows this situation at a high level.

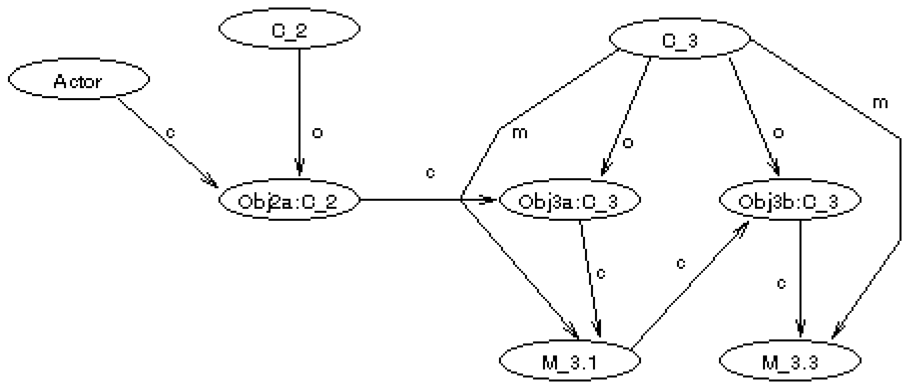


Figure 6.17: Model View of Figure 6.16. It explicitly includes the invocation relationship between an object and a method, and the calling relationship of one method to another.

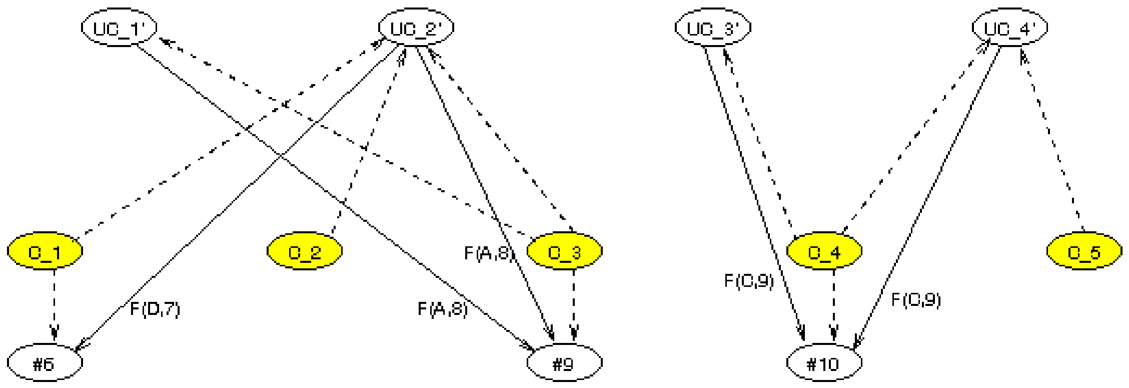


Figure 6.18: Longitudinal delocalization of annotations. Here, a viewer of specification $\breve{R}'$ sees annotations made on the classes in design $\breve{D}$ that together implement each use case. When considered as foreign annotations, some annotations appear multiple times in the same View.

First, let View U be the high-level View of all four use cases in $\breve{R}$'. The steps in filter construction are as follows:

- Find the design classes that implement each use case. Since the implements link "i" runs from a design class to a use case, we need to follow the inverse of the implements link, which we call "i*". As in Chapter 5, we assume that the inverse link is defined for all links in the graph of part of the artifacture. Then the design classes that implement the use cases are those within a neighborhood of U of distance 1, restricted to edges that are of type “i*”. Call this View of the use cases and implementing design classes V. Then $V = N^{i*}_1(U)$.

- Now find the annotations on these design classes. These will be the annotations of distance 1 from any of the design classes, which are just the Items in $V - U$. Thus, the annotations of interest, together with the use cases and design classes, form the neighborhood $V \cup N^{\alpha}_1(V\text{-}U)$. But we want to see the annotations on the design classes in the context of the use cases, not in the context of the design classes themselves. Thus we must apply a filter that creates foreign annotations, which will attach the annotations on the design classes to the use cases, essentially giving us $U \cup N^{\alpha}_1(V\text{-}U)$, which is the same as $N^{\omega}_1(U)$, since it is the neighborhood of all foreign annotations of depth 1 on the use cases.

View U is the View of the unshaded Items in Figure 6.18. A given annotation can appear in two separate places as a foreign annotation. In this case, classes C_3 and C_4 each implemented two use cases, so their annotations became foreign annotations on two different use cases.

Although we could also address historical and lateral delocalization in this design, these are easier to illustrate after we introduce more products of development.

1.1.4 Design Ď’

Annotated design Ď is replaced by revised design Ď’, which serves as a final design, and is not further annotated. Classes and their attributes and methods that have been updated from the original design Ď are indicated with deltas (Δ) in Figure 6.19. Figure 6.19 is an updated detailed class diagram, and Figure 6.20 shows the model View of this diagram. Differences are confined to changes in some of the attributes and methods of the classes; no new classes, attributes, or methods were added, and no existing ones were taken away. Although nothing about this modified design to exemplifies our method of addressing clutter and delocalization, we nevertheless show this stage because it is an intermediate step in getting to the implementation, and because we must traverse $\check{D}$’ to get to delocalized specification or design annotations when looking at the implementation.

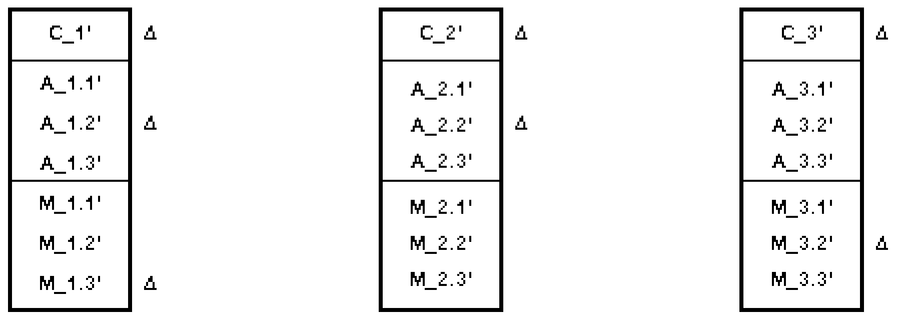


Figure 6.19: Updated detail of classes C_1, C_2, and C_3. Changed classes, attributes, and methods are denoted by (Δ). All items are primed (e.g., C_1’) to indicate the revised design.

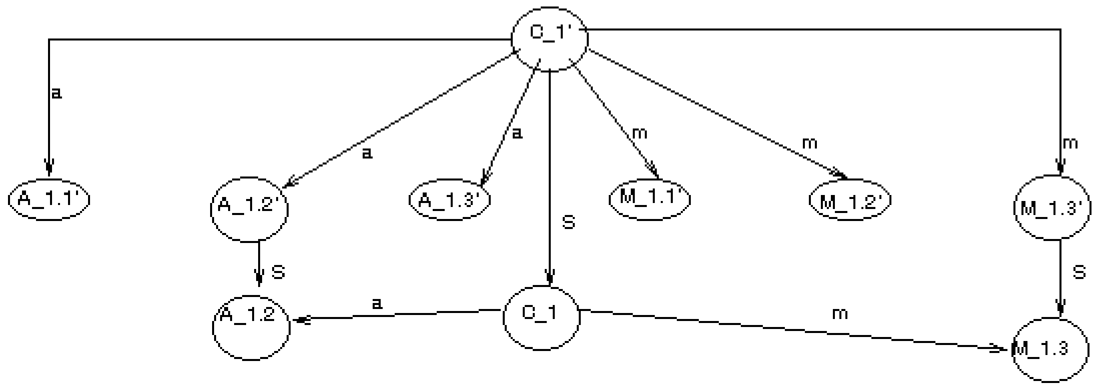


Figure 6.20: Model View of C_1’ from Figure 6.19. Note the historical delocalization of both the class and the changed attribute and method.

1.1.5 *Implementation $\tilde{I}$*

Figure 6.21 is a representation of annotated source code for one class in a Java-like programming language. Figure 6.22 shows the model View of this source code.

1.1.1.1 Addressing clutter and delocalization in Implementation $\tilde{I}$

Having shown two versions of the specification of this system, two versions of the design, and a version of the implementation, we can now demonstrate how to use our model to address complex delocalization problems. Specifically, we will use delocalization to show how a viewer could find annotations pertinent to the question: “Does method M3.3 of class C_3 implement use case UC_1 of the specification in such a way that it addresses the idea Bob proposed when reviewing the initial specification?”

A complex question such as this is a natural question in software development. There is a connection between the specification of a system and its implementation, and implementation of a certain portion of the specification can often be traced to a specific portion of the source code. When this traceability can be determined, it is reasonable to ask whether a certain portion of the implementation correctly implements a certain requirement from the specification.

```
public class C3  {

    <type> A3.1 = <initial_value>;
    <type> A3.2 = <initial_value>;  /* #14(D,13) */
    <type> A3.3 = <initial_value>;

    public <return> M3.1 ( <params> )  {   /* #13(C,13) */  // Method M3.1

        S1;
        <object>.M3.3( <params> ); // S2;  uses A_3.3
        S3;    /* #11(A,12) */
    }

    public <return> M3.2 ( <params> )  {     // Method M3.2

        S1;       // uses A3.2   /* #15(A,13) */
        S2;
        S3;       // uses A3.1
    }

    public <return> M3.3 ( <params> )  {   /* #12(B,12) */  // Method M3.3
    // This method is a key part of implementing use case 1 of the specification

        S2;
        return <value>;   // S3;
    }
}
```

Figure 6.21: Class C_3 from Implementation Ĩ. The Si are statements within the methods that we do not expand. The brackets enclose values that would be supplied in an actual

implementation. The “//” is the symbol for commentary. An additional commentary symbol is used for annotations. /* #11(A,12) */

This is a combined clutter and delocalization problem with all three types of delocalization. Clutter appears because we are only interested in Bob’s concern. Longitudinal delocalization appears because we want to see an annotation made on a specification from the point of view of the implementation. Historical delocalization appears because we must find the annotation on the reviewed version of the specification, not the implemented version. Finally, lateral delocalization appears because we have to look from the method to the class in the implementation to get the answer.

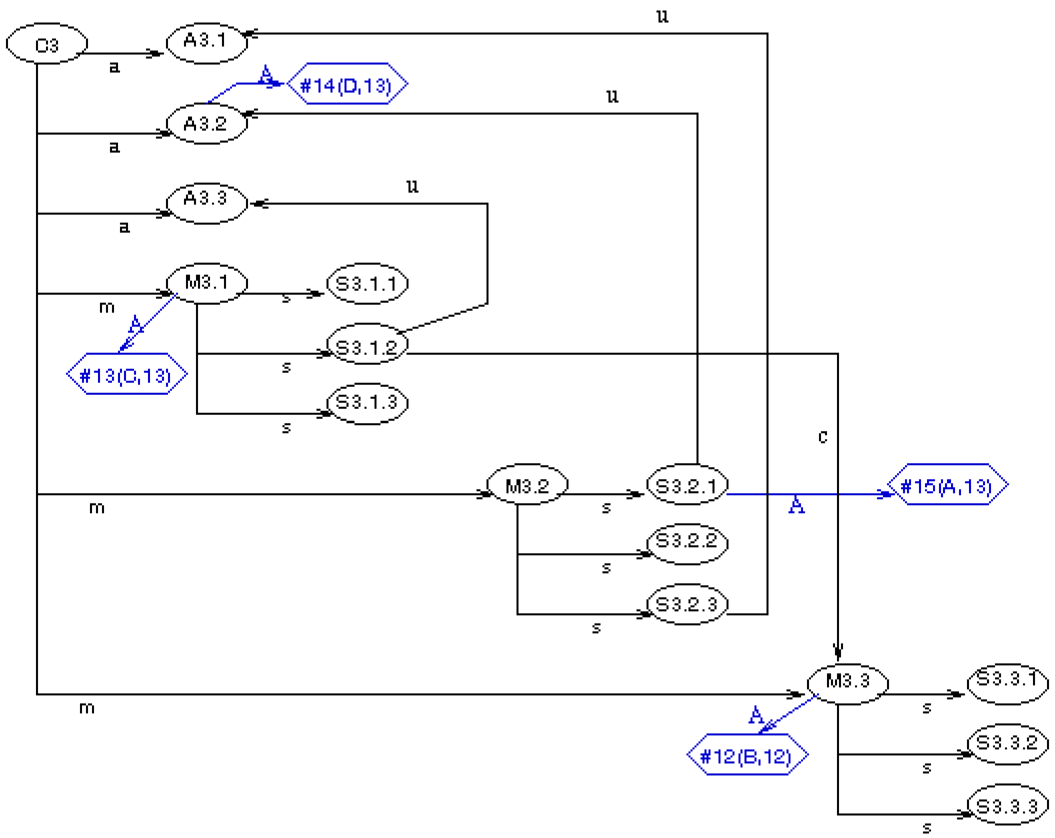


Figure 6.22: Model View of Figure 6.21. The uses and calls relationships among the statements of C3 are shown.

To address this problem, we take the following steps:

1. Determine the neighborhood of the method that includes its class. This will be the neighborhood of the method that contains all classes related to this method by the "has-method" relationship. This neighborhood of method M3.3 is $X_1 = N^{m^*}{}_1(M3.3)$. In words, this is the union of M3.3 with its class C_3.

2. Backtrack from the implementation of the class to the (revised) design of the class. In this case, we want to find the neighborhood of the implementation of class C_3 that contains its design. That is, we want to find class C_3 in the final design $\breve{D}'$. The neighborhood of X_1 containing that design class is $X_2 = N^{i^*}{}_1(X_1)$. This actually yields the final design class C_3' and the design method M3.3', plus all the Items of X_1.

3. Backtrack from the (revised) design of the class to the initial design of the class. That is, we want to apply the inverse of the supercedes link on X_2 to get the initial designs of class C_3 and method M3.3. This neighborhood is $X_3 = N^{s^*}{}_1(X_2)$. This gives us the initial design of both the class C_3 and the method M3.3, plus all the Items of X_2.

4. Identify the use case(s) that the class implements. To do this, we look for the neighborhood of those PIs that class C_3 implements. This neighborhood is found by using the inverse of the implements link to yield $X_4 = N^{i^*}{}_1(X_3)$. This will yield the use cases that the class implements, along with the Items of X_3. (Since the class as a whole implements a use case, carrying along a method in our computation causes no problems.)

5. Find the annotation(s) from Bob on those use cases, and attach them to the method. To do this, we find all annotations made or replied to by Bob on the neighborhood X_4. This yields $X_5 = N^{\alpha_{(B,*)}}{}_\infty(X_4)$. We then apply Corollary 7.2 of Chapter 5 to attach the annotations on the use cases to the method M3.3 of class C_3 in the implementation.

X_5 contains annotations made or replied to by Bob not only on use case 1, but also on use case 2, class 3, and method 3.3. These annotations might also be valuable. If not, we could do more filtering to subtract out unwanted parts of the neighborhood at each step of the computation.

1.1.6 UML Example summary

This example has shown how to use the graph-theoretic model of chapter 5 to compute visible annotations to address the clutter and delocalization issues for OO software artifacts. This example is important because OO artifacts form a large set of modern software artifacts.

1.2 General Textual Document Example

To illustrate the generality of the graph-theoretic model approach, we illustrate an application of the model to an entirely different type of artifact, a textual document with no formal structure beyond division into chapters and sections. Specifically, we consider how one might use the model to review a document such as this thesis minus its tables and figures. We discuss the nature of the PIs in this thesis, and we discuss how the model might be used to address problems of clutter and delocalization.

1.2.1 PIs of this Example

Assume that this document consists of text only. For this thesis, consider the atomic PIs of text to be *words,* and consider each term in a mathematical equation to be a word. Words are grouped into *paragraphs*. Paragraphs are grouped (optionally) into *sub-subsections* (level 4 headings). Sub-subsections are grouped into *subsections* (level 3 headings). Subsections are grouped into *sections* (level 2 headings). Sections are grouped into *chapters* (level 1 headings).

To complicate matters, not all paragraphs reside under sub-subsections. Many parts of the document have paragraphs directly under higher-level headings. For instance, this paragraph is directly under a subsection, and the first paragraph of section 6.2 ("To illustrate …") is directly under a section. Figure 6.23 illustrates the flow of text described above.

In addition to the hierarchical connections, Figure 6.23 shows connections among words. The main connection is "identity", in which all identical words are connected by the identity link, which is bi-directional. We assume that two words are the same if and only if all their letters are the same, regardless of capitalization. We further assume that a word and its plural are the same.

1.2.2 Addressing Clutter and Delocalization in General Text

Our model can be useful in the annotation of a document that has no well-defined formal structure, such as this thesis. This section gives some examples of how that can happen.

Filtering to reduce clutter would be similar to narrow-filtering as described in section 6.1. We could easily filter annotations based upon their author, their date, or a combination of both. We could also filter annotations by scope, for instance we could narow-filter to obtain only annotations of subsection-level or higher scope. Additionally, we could have annotators classify the annotations they make by type, so that viewers could narrow-filter annotations by type (e.g., to show all annotations indicating grammatical errors).

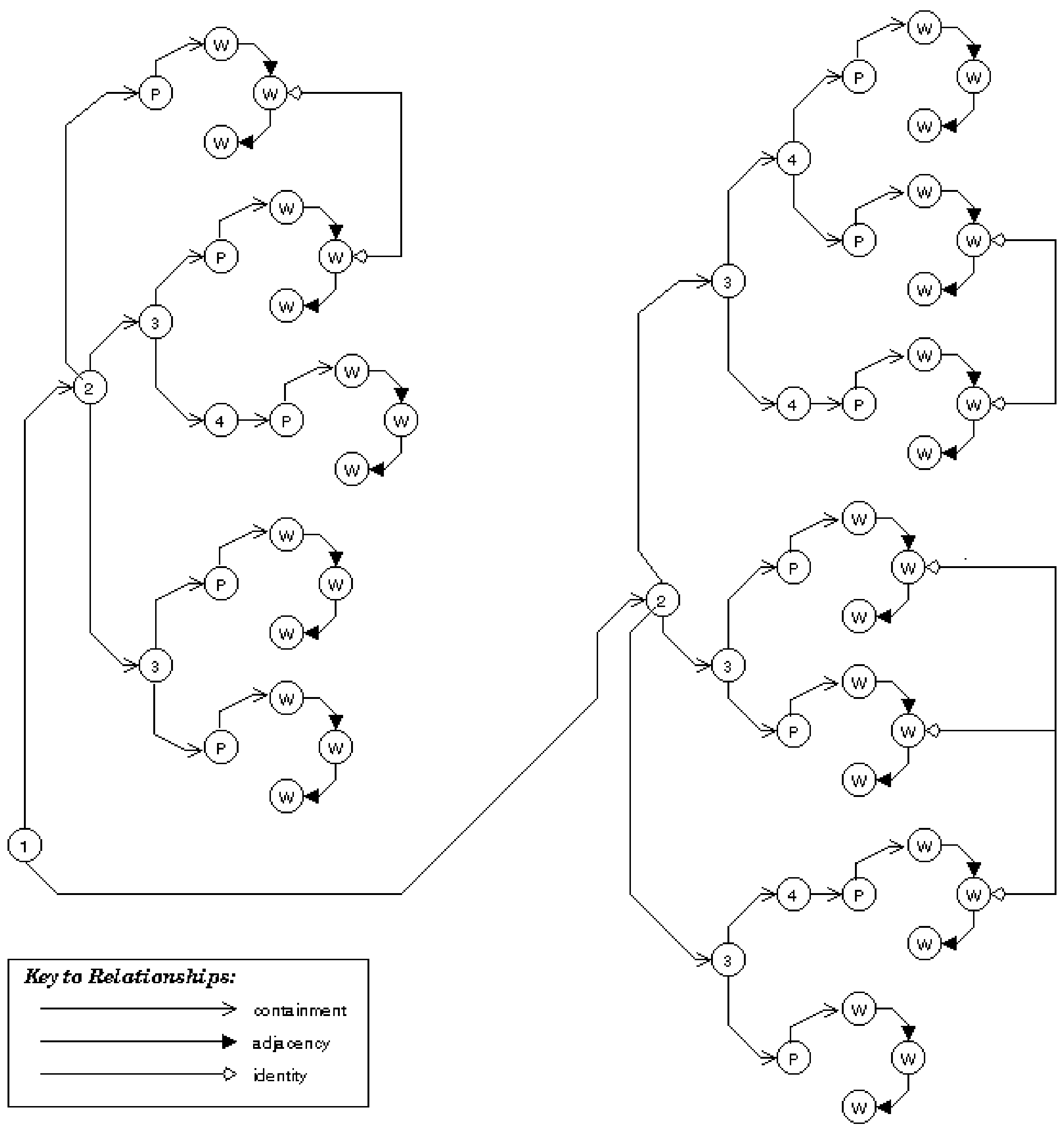


Figure 6.23: Connections among Items in general text. Connections shown are inclusion among section headings, paragraphs, and words ("containment"); word order from paragraph to first word and from one word to the next ("adjacency"); and identity of different words ("identity").

A single textual document that is not one stage of a set of documents (such as a software design document) would not have longitudinal delocalization. Historical delocalization between successive drafts would still exist, as would lateral delocalization between the same concept in different parts of the thesis. Let W_0 be the word "Item". Suppose that a viewer is looking at the final draft of the thesis, sees the first use of W_0 in chapter 1, and decides to see if other viewers have had any comments on the concept in any draft of the document. To find the desired annotations, this user must address both lateral and historical delocalization.

Let i be the identity link among different uses of the same word, let s be the supercedes link, let $c(x)$ be the containment link indicating containment of Item x, and let t_k be the k-th (and most recent) version of the document. Then the viewer might start by getting all versions of the document by applying a filter to get neighborhood $X_1 = N^s{}_\infty(t_k)$. X_1 is the set of all versions of the document.

Then the viewer might obtain all instances of the word W_0 by applying the containment filter $c(W_0)$ filter to obtain the neighborhood $X_2 = N^{c(W0)}{}_1(X_1)$. We only need a neighborhood of depth 1, because we assume that all copies of the same word are interconnected directly. To find the annotations on all these occurrences, the viewer now applies a filter to yield the annotations, getting $X_3 = N^\alpha{}_1(X_2)$. Finally, the viewer follows the procedure of Chapter 5, Axiom 4 to use foreign annotation edges to connect the annotations in X_3, that is the set $X_3 - X_2$, to the instance of word W_0 being viewed.

1.3 AISA Example

AISA, described in Chapter 2, allows users to inspect generalized files in three phases: fault collection, fault correlation, and fault resolution. AISA behaves differently in each phase. AISA classifies each user as a Producer, Moderator, or Reviewer; the actions of a user in a phase are governed by user role. We discuss the use of AISA and its relationship to our model by phase.

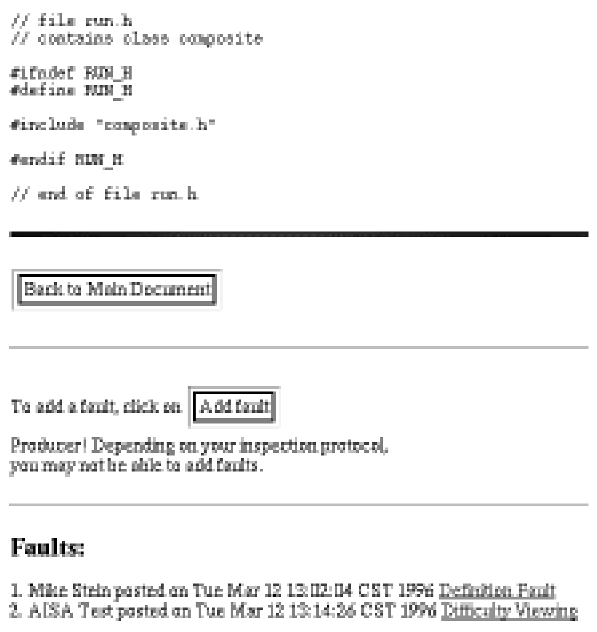
1.3.1 Fault Collection

In fault collection, a user logs in and selects a file. AISA shows the user a screen containing the artifact at the top and the set of annotations made on that artifact at the bottom, with navigational information between (Figure 6.24). The file can be textual or graphical. An annotation is added to the file by pressing the “Add Fault” button in the middle of the page. Every annotation has a scope of the entire file, because there is no way to attach an annotation to a given part of a file. The annotations are shown, numbered, at the bottom of the page. The date, author, and elided title, which compose the kernel of the annotation, are shown for each annotation. Selecting an elided title gives the entire contents of the annotation. Replies to annotations are not permitted in the fault collection phase of inspection. Producers are not permitted to annotate artifacts.

In our model, each file is effectively a separate atomic Item. Call these AISA Items $I_1, \ldots, I_n$. The set of all files may be considered a composite Item. Each production View of the artifact consists of exactly one Item. Thus we can define Views $V_i = I_i$, $i = 1, \ldots, n$. Each production View Vi has associated with it an annotated View V_i', containing all the annotations associated with that Item/View. The annotated Views are given by $V_i' = N^{\alpha}_{1}(V_i)$. No filtering is permitted in these Views, and all users see all the annotations.

1.3.2 Fault Correlation

In the fault correlation phase, the Moderator creates a master fault list in a two-step process. First, the Moderator sees a numbered list of all annotations on a given file, and can use a form to delete annotations by number or merge multiple annotations by number into one annotation. This sub-phase of fault correlation is called “local fault correlation”. (Figure 6.25)



```
// file run.h
// contains class composite

#ifndef RUN_H
#define RUN_H

#include "composite.h"

#endif RUN_H

// end of file run.h
```

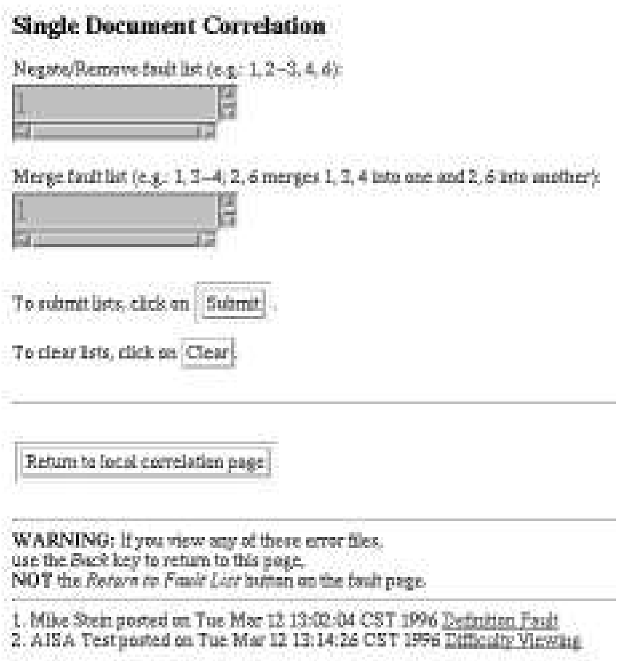
To add a fault, click on

Producer! Depending on your inspection protocol,
you may not be able to add faults.

Faults:

1. Mike Stein posted on Tue Mar 12 13:02:04 CST 1996 Definition Fault
2. AISA Test posted on Tue Mar 12 13:14:26 CST 1996 Difficulty Viewing

Figure 6.24: AISA Fault Collection Phase



Single Document Correlation

Negate/Remove fault list (e.g.: 1, 2–3, 4, 6):

1

Merge fault list (e.g.: 1, 3–4; 2, 6 merges 1, 3, 4 into one and 2, 6 into another):

1

To submit lists, click on.

To clear lists, click on.

WARNING: If you view any of these error files,
use the *Back* key to return to this page,
NOT the *Return to Fault List* button on the fault page.

1. Mike Stein posted on Tue Mar 12 13:02:04 CST 1996 Definition Fault
2. AISA Test posted on Tue Mar 12 13:14:26 CST 1996 Difficulty Viewing

Figure 6.25: Fault correlation in AISA. This figure shows a view of local correlation, in which the moderator can select which annotations on a given file to combine, and which to delete.

After performing local fault correlation, the Moderator performs "global fault correlation", by viewing a single numbered list of all annotations across all files, and merging and/or deleting annotations as in local correlation. The output of global correlation is a single numbered list of all the annotations on the files. Once fault correlation begins, nobody in any role can add any more annotations to the artifacts.

In the language of our model, during local correlation of a View V_i, the Moderator sees an annotation View of the annotations on V_i , namely $V_i' - V_i$. The Moderator manually provides a custom narrowing filter that eliminates some annotations and merges others, essentially creating a new file with a sort of subset of the annotations. It is not a true subset because if multiple annotations are merged, the merged annotation gets the author and date of the earliest of the merged annotations, with the title and contents of the other annotations concatenated to the contents of the earliest of the merged annotations. So a merged annotation is really a new annotation, since it is not identical to any previous annotation.

Let $A_{i1}, \ldots, A_{im}$ be the m annotations on View V_i. These annotations make up View $V_i' - V_i$. The Moderator manually filters these annotations to produce another set of annotations, call them $B_{i1}, \ldots, B_{ip}$, $p \leq m$, such that each of the B_{ik} either is the same as one of the A_{ik}, or is the union of two or more of those A_{ik}. Denote by W_i the annotation View of View V_i after local correlation.

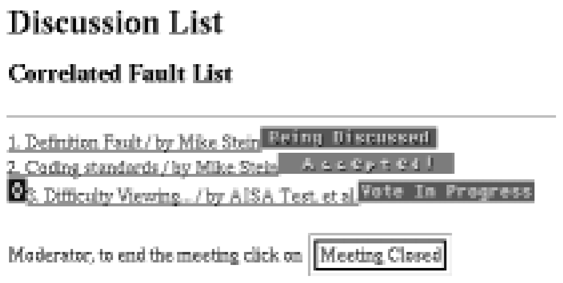
During global correlation, the Moderator is presented with a View consisting of the union of all the annotations from all the files after local correlation, $\cup_i W_i$, $i = 1, \ldots, n$. The Moderator deletes some annotations from this union and merges other annotations to come out with a set of annotations on the whole artifacture of the inspection. Essentially, the Moderator manually filters the annotations in $\cup_i W_i$ to produce a set of annotations $C_1, \ldots, C_r$ that are attached to the composite Item that is the artifact as a whole.

After fault correlation, the original artifacts still exist in the Views V_i, but they are available for historical purposes only and have no connection within the AISA system to the annotations C_j that are the output of fault correlation.

1.3.3 Fault Resolution

In the fault resolution (or discussion) phase, all participants see a single, global list of numbered faults as shown in Figure 6.26. Everyone can reply to these annotations and participate in a threaded discussion on each annotation. AISA also has the concept of "proposal", which is a special annotation that causes a vote to take place. If the vote is to "accept" the proposal, the annotation is placed in a "resolved" state in which no further additions may be made to its discussion thread. If the vote is to "reject" the proposal, the annotation remains "open" and threaded discussion continues. Various voting protocols may be defined in AISA, such as unanimous vote required to accept a proposal, or majority vote needed to accept a proposal.

While the voting takes place, threaded discussion may continue (earlier versions of AISA disallowed this, forcing a vote to be completed before anyone could add to the discussion). The Moderator may also at any time mark an annotation as "unresolvable", disallowing further additions to the threaded discussion.



Discussion List

Correlated Fault List

1. Definition Fault / by Mike Stein Being Discussed
2. Coding standards / by Mike Stein Accepted!
3. Difficulty Viewing... / by AISA Test, et al Vote In Progress

Moderator, to end the meeting click on Meeting Closed

Figure 6.26: AISA discussion phase. This shows all the annotations from all the files being inspected, but without any context —no indication is given which file was originally annotated. The diamond icon indicates a merged annotation.

During the discussion phase, when a viewer selects the hyperlink corresponding to an annotation, that viewer sees the complete annotation, and header information (title, author, date) for each thread in the threaded discussion.

The main View in the discussion phase is an annotation View consisting of the set of all the C_j as defined in the previous section. The kernel of each annotation is its author, date, and elided title. When a user selects the hyperlink corresponding to the title of an annotation, the user sees a new View, which is an annotation View consisting of the selected annotation and all its replies. The user sees the entire annotation, and the kernel of the replies.

1.3.4 Filtering of Annotations in AISA

AISA does not allow filtering of annotations. This section addresses the question of whether our model could be used as a template to permit filtering of annotations.

Our model could be used to guide narrow-filtering of annotations. Each annotation in AISA could be considered to be attached to its annotated PI by links typed along author and date axes, so the annotations could be narrow-filtered along these axes. One problem that arises is the treatment of merged annotations. Since merged annotations take the identity of a particular annotation, information within the graph-theoretic model of the inspection is lost during the merger. However, it would be possible to revise merged annotations so that they were presented with one author and date, but linked in the graph to all authors and dates of the merged annotations. In the case of globally merged annotations, each merged annotation would be linked as a foreign annotation to all Items to which any one of the constituent annotations were linked. Since the standard View of annotations during Fault Resolution is an annotation View out of context of the artifact, this would not affect the conduct of the review, but it would permit the visibility of more annotations when narrow-filtering by author or date. (Thus, if annotations from Ann and Bob were merged and the merged annotation had Ann’s name on it, our model could save the information that Bob was a co-author of the merged annotation, and when a user narrow-filtered the Discussion similar to Figure 6.26 to find Bob’s annotations, the merged annotation would be visible.)

Because of the coarse granularity of PIs in AISA, and because in the Discussion view all annotations are shown by default, lateral broad-filtering would not make sense for AISA. Longitudinal broad-filtering might be practical on a limited scale, for instance broad-filtering between a design diagram and the source code files that make up the design elements defined on that design diagram. Historical broad-filtering would work as it would in the other examples, where annotations from the reviewed version of an artifact could be viewed by those looking at a later version of the same artifact.

1.3.5 Example of Theoretical Addition of Filtering to AISA

Consider the UML example of section 6.1 as it would be inspected under AISA. For this example we ignore the possibility of merging annotations, as that would obscure the main point.

Suppose that the specification of $\check{R}$ was contained in four files, one for each use case and one containing the relationships among the use cases; denote this latter file "UCs". We use the "part of" link defined in Table 6.1 to indicate that the high-level figure contains the three use cases. Figure 6.27 shows the four files, and indicates which annotations would be native annotations on each file. Since the Items are Views, an annotation can be native in only one View. Also, there is no difference in annotation scope between an annotation on an entire use case and an annotation on part of it. Thus annotations #2 and #5 both annotate UC_2, and will be inseparable under any manipulations of annotations (except filtering by author, date, or other annotation-specific property).

Figure 6.28 shows the Views in updated specification $\check{R}$'. In this case, there are nine separate Views, one for each of the three use cases from $\check{R}$, one for each of the four use cases from $\check{R}$', and one for each of the high-level Views in each specification. Suppose that someone looking at the high-level use case diagram *UCs'* in the updated specification wants to see the annotations made on the original use cases.

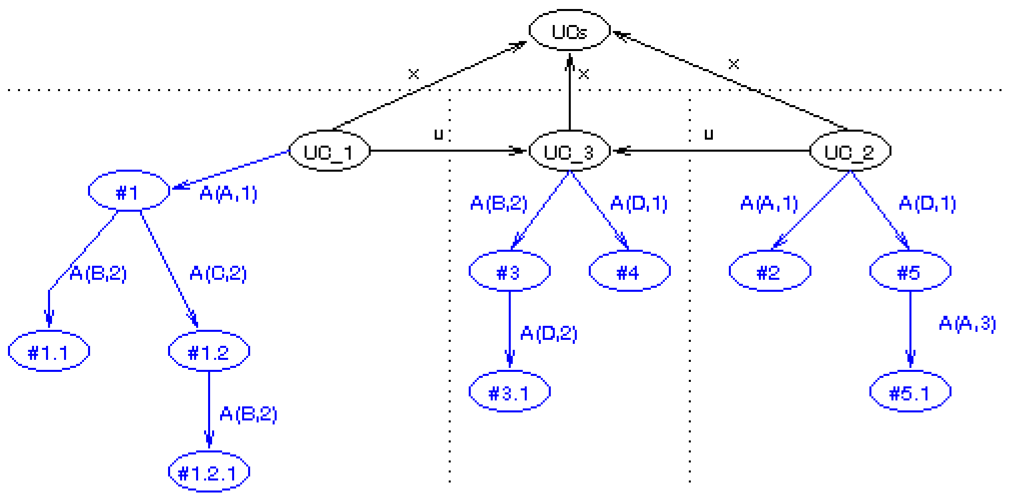


Figure 6.27: Model View of specification Ř under AISA. Each use case is "part of" the master use case View. Boundaries of the four files are shown by dotted lines.

One way to do this would be to start with Item/View *UCs'*, and to find the high-level use-case diagram that this supercedes. The neighborhood to be found by the filter is $X_1 = N^S_\infty(UCs')$, which is just $UCs' \cup UCs$. From here, we find all use cases in the original specification by applying the inverse part-of link to get the individual use cases. Call this $X_2 = N^{x*}_1(X_1)$. X_2 includes the original use cases. Then we find the annotations on these original use cases (and on the original high-level diagram, which happens to be unannotated) by defining neighborhood $X_3 = N^\alpha_1(X_2)$. Finally, we apply the construction of Corollary 7.2, Chapter 5 to attach foreign annotations to *UCs'*. The result is shown in Figure 6.29.

In Figure 6.29 there is no context available for the annotations. It cannot be determined directly by a viewer of *UCs'* which original use case was being annotated by which annotation. Thus, the ability to view foreign annotations loses some of its power, because the annotation may not say which Item it refers to. An annotation such as "completely rewrite this use case" is meaningless as a foreign annotation unless the annotation can be associated with a specific use case.

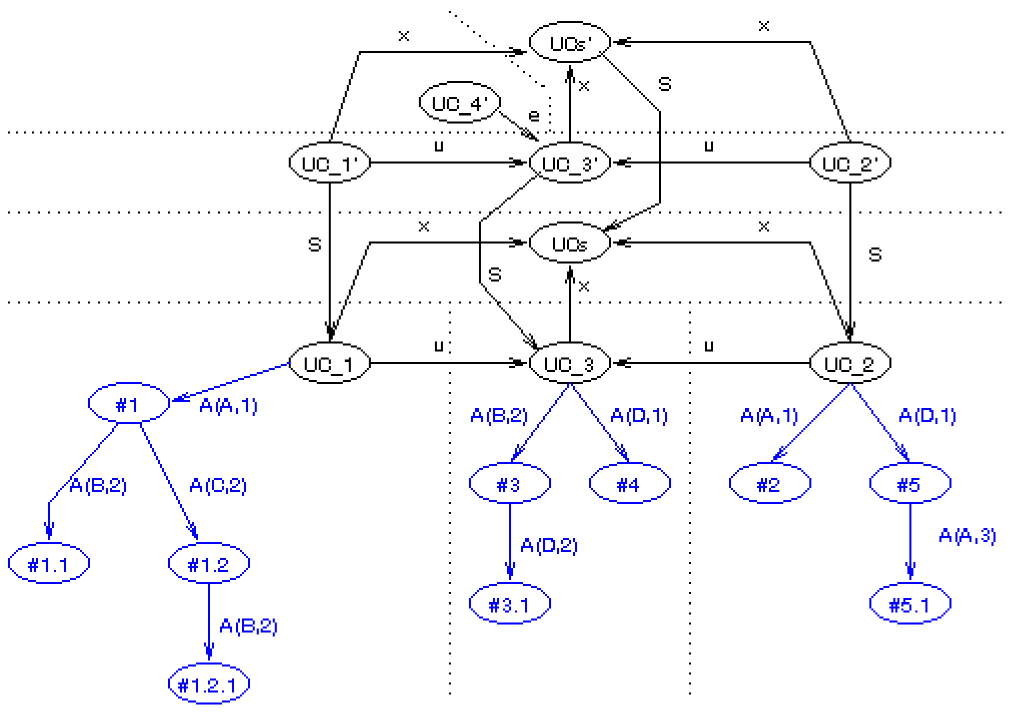


Figure 6.28: Historical delocalization in AISA. This figure shows all the relationships among the Items/Views of both versions of the specification in AISA, and the annotations on the original use cases. Again, the dotted lines delineate the Views.

In summary, it is possible to use our model to describe annotation visibility within our research group's previous inspection tool, AISA. We have further shown how delocalization of annotations could be addressed within AISA, although we noted that AISA is not designed to effectively show foreign annotations, because of the course granularity with which AISA attaches annotations to artifacts.

1.4 CSRS/FTArm Example

CSRS is a framework for distributed, asynchronous software inspection. Within this framework, specific tools can be defined to perform inspections under various protocols. The standard protocol used with CSRS is called FTArm (Johnson et. al., 1993). This section describes how our model can be applied to inspections conducted with CSRS/FTArm.

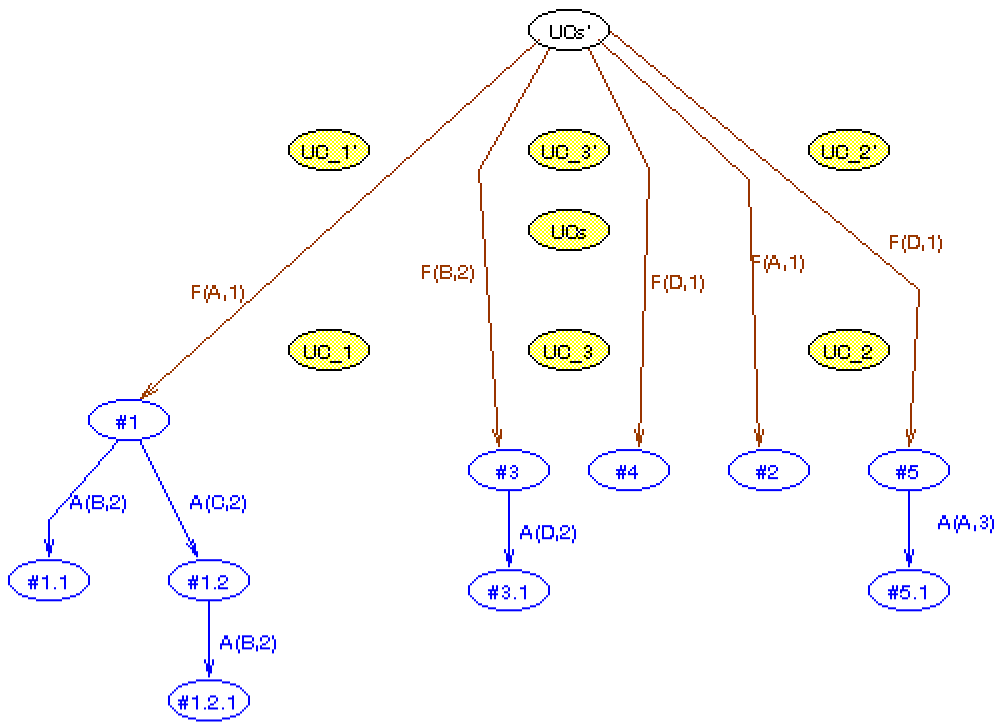


Figure 6.29: Foreign annotations in AISA. Here we show the foreign annotation connections from Item/View *UCs'* of specification *Ř'* to the annotations made on the use cases in specification *Ř*. The intermediate edges are hidden to enhance clarity.

1.1.1 Overview of CSRS/FTArm

CSRS/FT Arm uses an inspection protocol similar to AnnoSpec and AISA. Each inspection has a Moderator, a Producer, and one or more Reviewers. The artifact is divided into *nodes*. The *source nodes* are defined at the time the inspection is initialized, and the tool can be configured to provide nodes of types appropriate to the material being inspected. For instance, node types in C++ code might include class-declaration, member-function, template, member-variable, private-part, public-part, etc. Node types in a requirements specification might include overview, functional-requirement, performance-constraint, hardware-limitation, etc. Nodes can contain other nodes; thus, the private-part of a C++ class might contain member-variables.

The first phase of inspection is "private review", in which the Moderator and Reviewers can annotate nodes. Nodes are of three types: *issues*, *actions*, and *comments*. Issues and actions can only be seen by the annotator. Comments can be seen by all, and a threaded discussion may be carried out. Private review is roughly equivalent to our “Fault Collection” phase of Chapter 4, with the addition of comments as a way to ask for clarifications that can be seen by all. Reviewers must mark each source node as reviewed, even if they do not comment on it. The Producer can make comments, but not raise issues or actions. The Moderator cannot see the annotations of the other Reviewers, but can see which nodes each Reviewer has completed.

Private review is followed by “public review”, which is analogous to our discussion phase of Chapter 4. Public review permits voting as in AISA, except that whatever the outcome of the vote, discussion of an annotation may continue until the Moderator stops the public review phase and issues the inspection report.

In either review phase, CSRS/FTArm provides for threaded discussion of visible annotations. It provides separate windows for the source node and the visible annotations on that source node. It is unclear from the literature how replies to annotations are presented.

1.1.2 Applying our Model to CSRS/FTArm

Our model applies directly to CSRS/FTArm. Source nodes are PIs in our model. They can be atomic or composite PIs. For instance, in C++ code, a “public-part” source node would be a composite PI containing various public PIs within a class, such as the set of all public member functions. Issues, actions, and comments are all annotations. The default visibility during private review is to hide issues and actions from all users except the author, and to allow everyone to see comments. The default visibility during public review is to allow everyone to see all annotations. Annotations can be made having different scopes, because source nodes can contain other source nodes. In general, CSRS/FTArm can use the exact node structure that we have described in Chapter 5 and used for the first two examples in Chapter 6.

1.1.3 Filtering Annotations in CSRS/FTArm

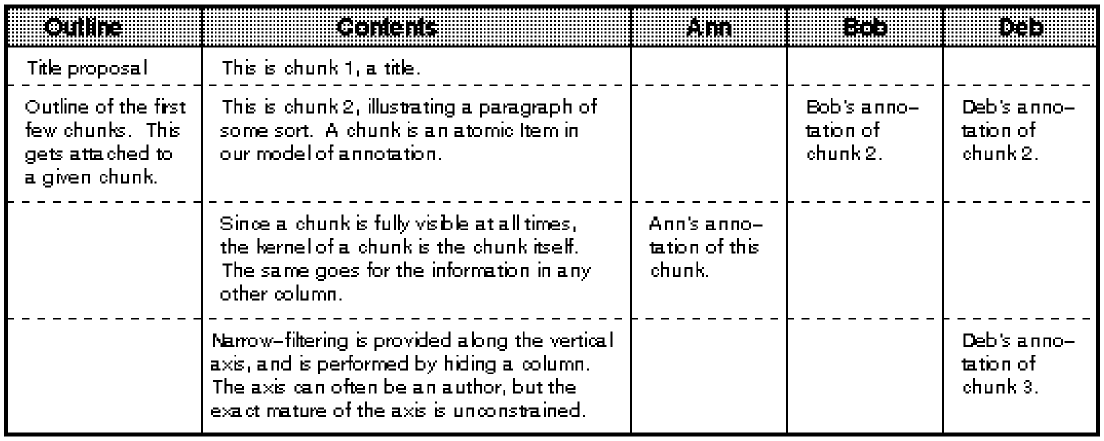
Filtering annotations using our graph-theoretic model is straightforward with CSRS/FTArm. Since we can map the Items created by this tool directly to our model, we can immediately use the full power of our model to narrow-filter and broad-filter annotations made on CSRS/FTArm inspections. Computing broad-filtered annotations in CSRS poses no theoretical problems in creating such an implementation or showing the foreign annotations to a viewer.

1.5 PREP Editor Example

The PREP editor is a collaborative annotation and authoring tool that uses columns to organize annotation information (Cavalier et. al., 1990). Each column serves a different purpose, such as showing an outline of a section, the document itself, annotations of a particular author, etc. There appear to be few formal constraints on what can be in a column. Although the PREP editor can be used for purposes besides annotating documents, but we are limiting our discussion to its use for annotations, since that is the scope of our model.

1.5.1 Basic Workings of PREP

One column is the document under study, which is divided up arbitrarily into “chunks”. Each chunk can be annotated separately. The scope of an annotation is the chunk to which it is attached. Figure 6.30 shows a stylized representation of this two-dimensional grid of the annotated document. Viewers have the option of showing whatever columns they wish, and of changing the widths of the various columns.



Outline	Contents	Ann	Bob	Deb
Title proposal	This is chunk 1, a title.			
Outline of the first few chunks. This gets attached to a given chunk.	This is chunk 2, illustrating a paragraph of some sort. A chunk is an atomic Item in our model of annotation.		Bob's anno-tation of chunk 2.	Deb's anno-tation of chunk 2.
	Since a chunk is fully visible at all times, the kernel of a chunk is the chunk itself. The same goes for the information in any other column.	Ann's anno-tation of this chunk.		
	Narrow-filtering is provided along the vertical axis, and is performed by hiding a column. The axis can often be an author, but the exact mature of the axis is unconstrained.			Deb's anno-tation of chunk 3.

Figure 6.30: Stylized representation of the PREP editor. This figure shows the layout of annotated documentation into multiple columns.

Users of PREP may reply to annotations by creating a new column for each person's replies. For instance, if Deb had a response to Ann's annotation of chunk 3 in Figure 6.30, she could create a new column for "Deb's responses to Ann", and place the reply there. In general, PREP is not meant to serve as a threaded discussion tool.

1.5.2 A Graph-theoretic Model of PREP

PREP is not an annotation tool, and it has a structure much different from the tools we have described previously. Therefore, to most effectively model PREP, we have to slightly modify the model to consider chunks and columns, not PIs and annotations.

In our model, the "contents" window contains the PIs of the artifact. A View consisting only of the contents is a production View of the documentation. If all annotation columns are shown, we do not filter annotations (we assume that the entire contents document can be shown by scrolling). This View of the document is the "standard" annotated View. Any column(s) can be filtered out of the View, including columns relating to the Items (e.g., contents and outline in Figure 6.30), annotations, or replies. Essentially, the content is a semantic issue set up by the administrator of the document.

When applying our model to PREP, we recognize that PREP Items are not semantically distinguished. They merely have chunk and column numbers. Thus we model PREP is as a grid of n chunks and m column, yielding a PREP document of $m*n$ Items. It is easiest to think of all these Items as existing all the time, just with some Items null. Then we supply an arbitrary total ordering of the columns, under which each Item has the following links to other Items:

- A "contains" relationship from the composite "chunk" Item to each Item in that chunk. Call this link *f*. This has an inverse "chunk-ID" link *f'*.
- A "contains" link from the composite "column" Item to each Item in that column. Call this link g. This has an inverse "column-ID" link, g'.

These links not only exist from an individual chunk or column to the atomic Items of that composite Item, but also from the document as a whole to the set of all columns and the set of all chunks. We parameterize these links from the document to its chunks and columns by chunk or column number. Figure 6.31 illustrates this set of links for the document in Figure 6.30.

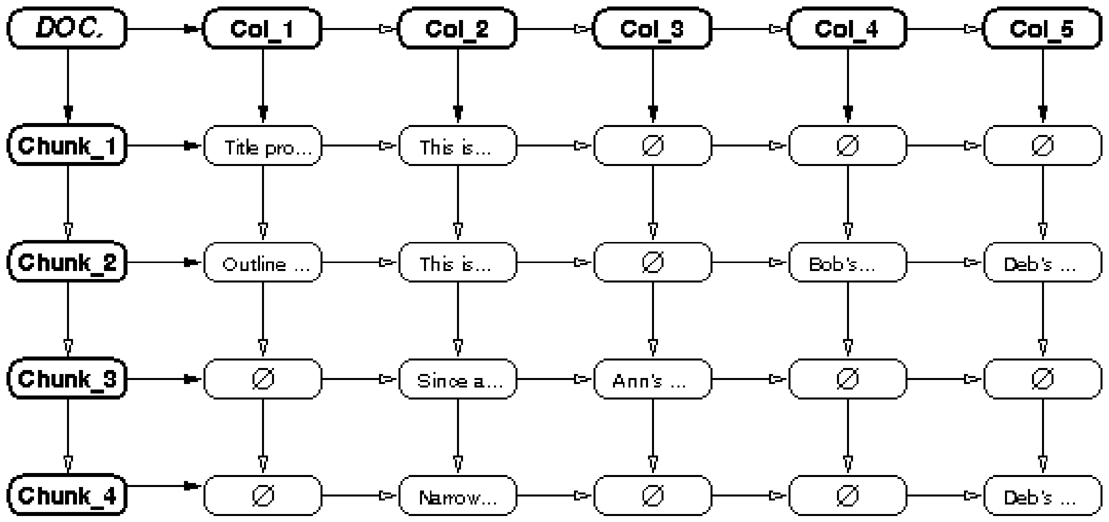


Figure 6.31: Links between chunks and columns in PREP. This shows the example of Figure 6.30. The closed arrows indicate the links to the first Items (those in column 1 and/or chunk 1). For simplicity, the other links are drawn as if from one Item to another, even

though they are links from a Chunk or Column to a specific Item. The document has the same relationship to its constituent chunks and columns as the chunks and columns have to their constituent Items.

1.5.3 Filtering Annotations in PREP

A simple way to filter is to start with the View V of the entire document. A user who wants to filter out column Col_n can merely subtract out Col_n from the View, so that she now sees the View $V - Col\_n$. Although the PREP editor has a window which allows users to see all chunks by scrolling, a user could conceptually filter out all but a certain chunk using this simple method.

We can define the graph-theoretic model for PREP as we did is that there are no cycles in the graph under the relationships f and g. Cycles could only occur when using forward and inverse relationships together.

As an example of narrow-filtering, suppose that column 1 of a document is an outline of that document, column 2 is the text of that document, and columns 3, 4, 5, and 6 are the annotations of Ann, Bob, Chris, and Deb, respectively. Then to see the outline, contents, and annotations of Ann, we compute the neighborhood $N^{f(1)}{}_1(DOC.) \cup N^{f(2)}{}_1(DOC.) \cup N^{f(3)}{}_1(DOC.)$. The depth of the neighborhood is unimportant because there are no secondary links. In essence, all neighborhoods are of depth 1.

PREP elegantly supports two axes for filtering. For any document D, we can apply a filter to see the set X of all chunks in a given column x of D, $X = N^{f(x)}{}_1(D)$; we could of course take the union of multiple columns. We could also apply a filter to see the set Y of all columns in a given chunk y of D, $Y = N^{g(y)}{}_1(D)$; again, we could take the union of multiple chunks. Finally, we could compose these filters to see a given chunk of a given column either by applying $N^{f(x)}{}_1(Y)$ or $Y = N^{g(y)}{}_1(X)$. Again, we could apply this filter to multiple chunks or columns, respectively, and we could union sets of column-chunk pairs to get any desired entry or entries in PREP.

1.6 Summary

In this chapter we have used our mathematical model of annotation to discuss, and in some cases compute, annotation visibility for five examples.

The first, most detailed example, was an OO software system represented by a specification, a design, and code. Using this system, we illustrated various ways in which a viewer could narrow-filter and broad-filter annotations to see desired annotations connected to that viewer’s present view of the document.

The second example showed how our model can be applied to a general textual document. This example illustrates that our ideas are not limited to certain specific types of software artifacts; instead they are applicable to a wide range of artifacts, even general textual documents.

The third and fourth examples show how the model can describe artifacts inspected by our AISA software inspection tool and the CSRS software inspection tool, respectively. The fifth example shows how the model can describe documents being authored using the collaborative PREP editor. These examples indicate that our model is able to model the workings of real tools.

This section demonstrates how the model can theoretically address clutter and delocalization, and how it can model actual tools. These demonstrations show the power of our model. The next two chapters discuss an annotation tool incorporating narrow- and broad-filtering and the results of a pilot study of that tool. Combined with this chapter, they demonstrate that clutter and delocalization problems for annotation software artifacts can be successfully addressed.

CHAPTER 7: IMPLEMENTATION ISSUES

In Chapter 5, we discussed how to express delocalization relationships using a graph-theoretic model. In Chapter 6, we showed how the model can be used to find the desired annotations to address clutter and delocalization problems in OO artifacts and in general textual documents. We also showed how our model can express annotation visibility in some existing inspection and collaborative authoring tools.

As we argued in Chapter 1, a second facet to demonstrating the usability of the model and its underlying idea of foreign annotations is to build a tool that supports the visibility of foreign annotations, and to assess its usefulness. This chapter discusses the issues in building such a tool, which we call *AnnoSpec*, for *Anno*tation and in*Spec*tion. We discuss requirements for the AnnoSpec tool, its design, and its implementation. Finally, we describe the use of AnnoSpec in its present form, which was used for the pilot studies of Chapter 8.

7.1 High-level Goals for a Framework to Filter Annotations

To effectively address the clutter and delocalization problems, both as part of software inspection and more generally, we need to develop an annotation framework that will enable users to achieve seven goals. Once we have developed that framework, we can implement it to allow us to examine some hypotheses we have formulated about annotation of software systems. The goals for our framework are:

A. **Basic Visibility:** View annotated artifacts and make annotations to artifacts, maintaining the functionality of existing annotation and software inspection tools.

B. **Narrow-filtering:** View only a subset of the annotations of visible documents.

C. **Broad-filtering:** View annotations on Items related to an Item being viewed, using relationships among the Items to decide which other annotations to view.

D. **Combination filtering:** View a subset of the annotations on visible items, and on other Items related to visible Items. This requirement, a simultaneous application of narrow-filtering and broad-filtering, is the core of our framework.

E. **Formal Inspection Support:** Support a protocol for distributed, asynchronous software inspection.

F. **Rapid, Easy Development:** The initial version of the AnnoSpec tool should be specified for easy development, by limiting the scope of the tool.

G. **Ease of Use:** The AnnoSpec tool should be easy to use by software professionals and computer science students with little instruction.

The Broad-filtering goal [C], and its extension in the Combination-filtering goal [D], are not implemented by any tools of which we are aware. Many annotation tools meet the other goals.

7.2 AnnoSpec Requirements

We have placed the following requirements on the AnnoSpec tool. These requirements derive from the goals of our framework and our previous experience with software inspection. If a requirement is clearly meant to address a given goal, we indicate the goal [in brackets].

1. *Permit viewers to easily view an artifact and its annotations.* This is the crux of the Basic Visibility requirement [A]. Some sub-requirements of this requirement are:

   1.1. *Support textual artifacts.* We support only textual artifacts at this time for simplicity of initial tool development. We further treat these artifacts as being composed of well-defined physical lines. [F]

   1.2. *Make the tool user-friendly.* The tool should be user-friendly so that users will not be distracted by problems with the tool. Such lack of distractions allows users to concentrate on annotating software artifacts. Wide availability could increase the use of the tool, and give us more ability to assess its effectiveness. [G]

1.3. *Allow people to see updates on the annotated artifact easily.* Viewers of an artifact should be able to see whether annotations have been made to the artifact since they last viewed it, and if so where. They should be able to do this immediately upon logging into an annotation session on AnnoSpec, without having to navigate to each artifact in the set of artifacts under annotation to see whether any changes have been made. When possible, this information should be refreshable while viewers are on-line. [G][A]

1.4. *Make the tool platform-independent.* The tool should be usable by a variety of people in various situations. Users should be able to work anywhere they have a computer. As discussed in section 4.3, sometimes reviewers of an artifact will come from two or more organizations. It is likely that these organizations will use different sorts of computer systems. Platform-independence of AnnoSpec is useful for allowing all such people to work together on the same artifact. [G][A]

1.5. *Implement narrow-filtering to reduce clutter.* This requirement directly addresses the Narrow-filtering goal. [B]

1.6. *Implement broad-filtering to ameliorate delocalization by showing foreign annotations within an artifact, while still addressing clutter.* This requirement directly addresses the Broad-filtering goal. [C][D]

2. *Permit viewers to annotate an artifact.* When permitting viewers to annotate an artifact, we must allow them to make sufficiently robust annotations, indicating such information as the name of the author and the time of annotation. We also permit annotators to select a type for the annotation compatible with the artifact being reviewed, as is common in inspection tools (Knight and Myers, 1993; Mashayekhi, 1995; Stein et. al., 1997). [A]

2.1. *Allow viewers to annotate lines.* For simplicity of first implementation, the smallest scope of annotation that we support will be a line. [F]

2.2. *Allow viewers to annotate "logical" Items.* Such Items include paragraphs of text, or OO constructs such as classes, attributes, and methods. We concentrate on OO textual artifacts to maximize the variety of artifacts we can annotate with minimal development effort. [A][F]

3. *Support distributed, asynchronous software inspection.* Distributed, asynchronous software inspection is a formal procedure with which we have experience (Mashayekhi et. al., 1994; Stein et. al., 1997). Because it is a structured procedure, we can learn how well the tool works by comparing the effectiveness of inspection using AnnoSpec to the effectiveness of previous inspections we have conducted with other tools. Such a comparison will also, hopefully, illuminate the usability of our model. [E]

3.1. *Implement the inspection protocol described in Chapter 4.* As discussed in Chapters 2 and 4, we have previously found this protocol to be appropriate for distributed, asynchronous software inspection. [E]

3.2. *Support notions of inspection states and participant roles.* These notions are vital to effectively implementing our protocol. [E]

Table 7.1 maps each requirement to the goals it expresses.

Table 7.1: Mapping between annotation framework goals and AnnoSpec requirements.

Reqs.(↓) & Goals(→)	A	B	C	D	E	F	G
1 (general)	•						
1.1						•	
1.2	•						•
1.3	•						•
1.4	•						•
1.5		•					
1.6			•	•			
2 (general)	•						
2.1						•	
2.2	•					•	
3 (general)					•		
3.1					•		
3.2					•		

In fulfilling the above set of requirements, we will be addressing two of the issues of Neuwirth, et. al. (1998) for improved asynchronous communication tools. First, we have a "task-tailorable" representation because AnnoSpec will be able to handle different sorts of specific delocalization relationships for different artifacts. Second, we implement "asynchronous awareness" by allowing people to see which artifacts in the artifacture have been recently annotated, and by updating this information throughout an annotation session. Although not implemented at present, we should also be able to eventually allow people to have some control over visibility of annotations they make (e.g., some annotations might be intended for the authors only), thus addressing their "emergent sharing" and "public/private elements in a layout" issues.

7.3 Design

This section discusses some of the design choices we made concerning the overall system architecture, the choice of languages and database system, and the user interface. It also describes the division of AnnoSpec into modules. We discuss the architecture first, because it is the highest-level portion of the design. Then we discuss our choice of database, because that drove many other decisions. Within each section we first describe the design element that we chose, then we discuss alternatives that we considered and rejected.

7.3.1 Architecture

We developed AnnoSpec for use with standard Web browsers (Netscape Navigator and Microsoft Explorer), because they are widely available and have a familiar interface. We chose to present the annotated artifacts to viewers/annotators in HTML, since all browsers can exhibit HTML. We assumed that users would have access to browsers with the subset of features common to Netscape Navigator 4.0 and Internet Explorer 4.0, because most people at the time of our pilot study used either Navigator or Explorer for a Web browser, and because people can freely obtain updates to these versions of the above browsers. A primary advantage of HTML is that it is standardized, and transmissible as ASCII text. A disadvantage is that different browsers will use the same HTML to produce different-looking pages.

We chose to place the major functionality of AnnoSpec on a central server, which we refer to as the "AnnoSpec server" to differentiate it from the Web server that serves Web pages to clients. We chose such a "thin-client" design, with most of the functionality on the AnnoSpec server, for high performance on a variety of client systems. Our server configuration is shown in Figure 7.1. It consists of the following components:

- A Common Gateway Interface (CGI) script to interface between the client and the main program.
- A main program the accepts requests from the users, accesses the database as appropriate to meet the needs of the users, then creates an HTML document to send back to the users.

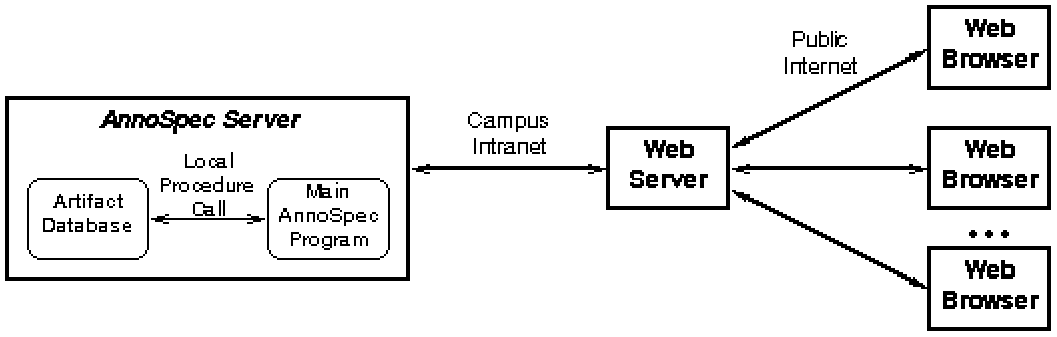


Figure 7.1: High-level architecture of AnnoSpec. The database and the main AnnoSpec program reside on the same machine for performance reasons. The AnnoSpec server connects to the clients through the Web server, which is a separate machine because of the constraints of the computing systems at the University of Minnesota.

- A database that holds the artifact and its annotations. The database resides on the AnnoSpec server for performance reasons.

Because of the configuration of the Web servers at the University of Minnesota, the Web server was a physically separate machine from the AnnoSpec server. The AnnoSpec server was connected to the web server through the standard University intranet connection.

7.3.1.1 Architectural Alternatives

One alternative that we seriously considered was to have users run AnnoSpec as a Java applet on their systems. This architecture has in its favor the advantage of browser-independence, and a Java applet gives us more control over exactly what the user sees than does an HTML page. However, sending a full applet to a user instead of just a page of HTML would cause performance bottlenecks, choosing to use applets locks us into using Java, and we felt that the development effort for using an applet architecture would be higher than for using standard HTML. Thus we did not choose this alternative.

Another alternative would have been to transmit an entire copy of the inspection to a user when they logged on, including a copy of the database, and let them work locally until they logged off. At logout we could have merged their new annotations back into a master copy of the inspection on the central server. This would have improved performance, but it would have required an AnnoSpec user to have the PSE database running locally. Moreover, people would not be able to see updates that other people had made without logging off and logging on again, unless we went to extra work to provide an update capability. However this approach causes no database inconsistency problems, since all annotations are additions to the database, annotations are never deleted or modified, and neither exact time nor order of annotation submission are critical.

7.3.2 Database System

As part of our architecture, we needed some place to store the artifact and inspection data. In other words, we needed some sort of database. We did not need to perform many complex queries of the database, but we wanted a simple database structure with good performance at a low cost, with a high-quality API (Application Programming Interface).

We chose to use ObjectStore PSE (Object Design), an object-oriented database, as our underlying database. PSE resided on the same server as the main AnnoSpec program. In our model of Chapter 5, the Items of an artifact form a tree under a subset of the links among them. PSE let us use this tree structure in the database, and let us manipulate attachments as needed. PSE also had a well-documented Java language API, which seemed to work well when we tested it. Finally, a single-user version of PSE is freely available to anyone (although it is limited to the Unix and Windows platforms). One disadvantage is that PSE would not be expected to scale well, as it is limited to relatively small (10s of megabytes) datasets. Were we to use AnnoSpec for annotation of a larger artifact, we would need to either upgrade to a costly ObjectStore product, or change to an entirely different database.

7.3.2.1 Database Alternatives

Alternatives to our using an object-oriented database would be to just use text files to store data and not use a database system at all, or to use a relational database.

AISA did not use a database system at all. It just stored all its data as ASCII text files on the server. The code had embedded within it a naming convention for these files and the directory structure of these files, which made the lack of a database system acceptable. However, this also made the system confusing because of the explosion of files and directories during a software inspection. Additionally, the code was confusing and the design unclean in order to support this file system manipulation. The advantage of this system was that it made AISA more portable because no separate database was needed to run the system.

Another possibility we considered was the use of a relational database, such as Oracle. Such databases are widely used in industry and academia, are robust, and are widely available. They also have powerful querying capabilities. However, they are best suited to situations in which users can define in advance the data they will be using and the relationships within the data, and want a powerful query capability. AnnoSpec merely needs a place to store data, and needs to be able to dynamically change the relationships between various of the data items (i.e., the Items of an artifact). We didn't need the powerful querying capability, and we couldn't define the relationships within the data *a priori*. After talking with database experts, we concluded that using a relational database was probably not the right choice for AnnoSpec.

7.3.3 Choice of Interface Language

In designing the AnnoSpec system, we had to choose a language to use for the CGI script. We first implemented the CGI script in Perl 5.0, because it is a convenient language for writing CGI scripts, and we were able to find source code on the Internet for a script that did most of what we needed to do. Once the Perl script was working, we discovered that it was slow. So we duplicated the functionality of the Perl script in C++, which we felt (correctly) would speed it up We wrote a C++ script, debugged this new script, then used the C++ script exclusively.

7.3.3.1 Language Alternatives

Reasonable language choices for a CGI script would be C, C++, or Perl. Perl scripts are well-supported at the University, and they are the most common choice at our location for CGI scripts. The only reason not to use a Perl script would be for performance reasons. Given that we had performance concerns, C or C++ were the logical choices for a compiled CGI script, because they are the fastest high-level programming languages we use. C++ is cleaner than C, being object-oriented, so we chose C++.

7.3.4 Choice of Language for AnnoSpec

When choosing a programming language for the AnnoSpec system we had a number of considerations that led us to ultimately choose Java.

Our first consideration was staffing. Our goal was to hire an undergraduate to do much of the programming. This meant that we wanted to use a language that was popular, so we would have a wide range of potential programmers to choose from. It also meant that we wanted to choose a language that undergraduates would find useful to have on their resumes.

Our second, technical consideration was that the language be well-supported at the university, and that the language make revision of the AnnoSpec code easy when that became necessary.

Our ultimate decision was to write the main program in Java, for the following reasons.

1. **Purity:** As a relatively pure OO language, it is also easy to write a well-designed program in Java and to understand and maintain it.
2. **Portability:** Java is a widely available language that is well-supported at our University. One factor in our choice of Java is that we could use Sun's Java Development Kit (JDK). The JDK is not an especially good development environment, but it is available for free to anyone on a Windows, Unix, or Macintosh platform.

3. **Programmer interest:** At the time we were looking for programmers, Java was considered an exceptionally good language to have on a resume, and an easy one to pick up if one knew C++ (as the undergraduates at this institution do). Advertising that the work would be done in Java probably maximized our applicant pool for the position.

The disadvantages to Java are that it is not as fast as C++, and Input/Output (I/O) operations are not as simple. Further, Java is a newer language and there was the risk of our encountering language features that did not work as advertised.

Finally, our choice to use ObjectStore PSE instead of some other object-oriented database was based in part on its elegant Java API. Had we chosen some other language, we might have chosen some other database system. The inverse of the previous statement is also true; had we settled first on some other database system, we might have chosen a programming language for AnnoSpec to be compatible with the chosen database. Effectively, the choice of database and the choice of language for the main AnnoSpec system were closely related.

7.3.4.1 Language Alternatives

Our alternatives to the Java language for the main AnnoSpec program were C++ and Perl.

Were AnnoSpec written in C++, it would probably run faster than it does in Java. However, it would not be as portable. We would also have had to address the interface issues, for which C++ is not considered to be a great language. C++ is also not as "pure" an OO language as Java, and we believe it is harder for people to understand a C++ program with which they are unfamiliar than it is for them to understand the equivalent Java program. Finally, the major performance bottleneck was painting the buttons to the screen, which was a language-independent problem.

Perl is a useful language for the text manipulation sorts of things that we do. Our previous inspection tool, AISA, was written in Perl for this reason. However. Perl is an interpreted language, and so it would probably be slower than Java (which is partially compiled), and especially slower than C++. Additionally, Perl programs (such as the AISA program) can often be notoriously difficult to understand without intensive effort.

7.3.5 User Interface: Artifact Display

Having chosen to use HTML to present users with the tool on our Web page, we were slightly restricted in our choice of user interface design to interfaces that could be specified with HTML. We discuss our major choices in this section.

Our goals were to present an annotation in proximity to the Item it annotates and to make it easy for people to annotate and view line-oriented files. To do this, we chose a 6-column tabular layout, as shown in the following Table 7.2.

The first column contains a button that a viewer can use to annotate "logical Items" on that line, where a logical Item is a portion of the line that forms a semantic whole. The present version of AnnoSpec doesn't allow finer granularity than to annotate all logical Items on a given line. If there are no logical Items to annotate on a line, this column is left blank.

The second column contains a button that allows a viewer to see native annotations made on the logical Items on a given line. If there are no annotations on logical Items on this line, this column is left blank. The button itself indicates if the annotation is new.

The third column contains a button that a viewer can use to annotate the given line as a whole.

The fourth column contains a button that allows a viewer to see native annotations made on the given line as a whole. If there are no annotations on this line, this column is left blank. The button itself indicates if the annotation is new.

The fifth column contains a button that allows a viewer to see foreign annotations of interest to viewers of this line and its logical Items. If there are no such annotations, this column is left blank.

The sixth column contains the contents of the line.

Pastel background colors are used to differentiate further among these columns. The first two columns, dealing with logical Items, are shaded blue. Columns three and four, dealing with lines, are shaded cyan. Column five, dealing with foreign annotations, is shaded yellow.

Table 7.2: AnnoSpec Column Layout. This is the six-column layout for textual files to be inspected using AnnoSpec. Colors are used to also differentiate among the columns. Logical Item columns 1-2 have blue backgrounds, line Item columns 3-4 have cyan backgrounds, foreign annotation column 5 has a yellow background, and text column 6 has a white background.

Button to annotate logical Items	Button to view logical Item annotations	Button to annotate line	Button to view line annotations	Button to view foreign annotations	Text of line being viewed

The major problem with this display was that it took up a lot of "screen real estate", so that only the right 50-60% of the window contained the text of the line being viewed. Additionally, buttons were large vertically in some browser/operating system combinations, meaning that few lines were visible on the screen.

7.3.5.1 Alternatives for Artifact Display

Alternatively, we could have allowed people to define the scope of an annotation by highlighting just the contiguous text they wanted to annotate, as done by CritLink (Yee, 1998) and Third Voice (Third Voice). But with this alternative we would then have needed to parse the text into its logical Items to see what people were annotating. Two people could annotate slightly different sets of contiguous words and still be annotating the same logical Item, which would greately complicate the attachment of foreign annotations. However, it would have allowed us to avoid the multi-column layout, which would have made for a cleaner interface. For viewing,. we could have placed a marker inline marking the beginning of the annotation as is done by Third Voice, or we could have bracketed the annotation with markers as is done by CritLink..

Another alternative would have been to retain the six-column layout, but to replace the buttons with hyperlinks. In some tests we ran, this greatly reduced the amount of screen real estate taken up by the annotation and viewing columns. However, by the time we discovered the extent of the button problem we were too far along to change the interface.

7.3.6 User Interface: Numbers of Windows

We wanted to present users with enough windows so that they could simultaneously see different views that they might want to see together. Yet we wanted to avoid unnecessary proliferation of windows. We chose a two-window system. One window would show the artifact, so that it was always visible when writing or reading annotations. The other window would be used for making and viewing annotations and for performing other activities (e.g., Moderators changing the inspection state, or any user changing annotation visibility). The only problem with having only two windows was that someone could only see one annotation at a time, although they could manually add more windows to preserve annotations they wanted to keep visible.

7.3.6.1 Alternatives for Numbers of Windows

We could have used only one window. AISA had only one window, so to view an annotation and the annotated artifact at the same time one had to manually manipulate windows within the Web browser. Thus we clearly wanted to have separate windows for the artifact and for creating or viewing annotations.

We considered having separate windows for viewing and creating annotations. That way, a user could have a window containing an annotation while replying to that annotation in another window. But after showing our ideas informally to user interface experts, we decided that such a proliferation of windows would be confusing. Instead, we chose to have the annotation viewing window be the same as the annotation entry window, with the property that when an annotator replies to annotation, the annotation being replied to appears at the top of the annotation window.

We also considered the use of frames for showing multiple pieces of information in the same window, but rejected this as adding more complications to the development effort.

7.3.7 High-level Design of Main Program

This section presents the high-level design of the main AnnoSpec program, including the division of AnnoSpec into modules. Our presentation is built around a series of UML-like diagrams, mostly class diagrams.

7.3.7.1 Modules

AnnoSpec is divided into five modules. Figures 7.3 – 7.7 show the high-level class diagrams for the classes in each module. Figure 7.2 is a key to the diagrams. Within the class diagrams, we hide certain methods. Specifically, we assume that for each attribute *xxx* of type *t* in some class, there is an unshown public method for retrieving the value of that attribute, of the form:

```
public t getXxx()  {  return xxx;  }
```

For instance, class MessagePacket in module AnnoServer has a String attribute “annonum”. Thus, class MessagePacket has the following method:

```
public String getAnnonum()  {  return annonum;  }
```

The *AnnoServer* module contains the server-related information, including the code necessary to handle packets between AnnoSpec and the web server. It includes the Inspection class of the entire inspection for convenience, and because the Inspection class does the bulk of the initialization of inspection.

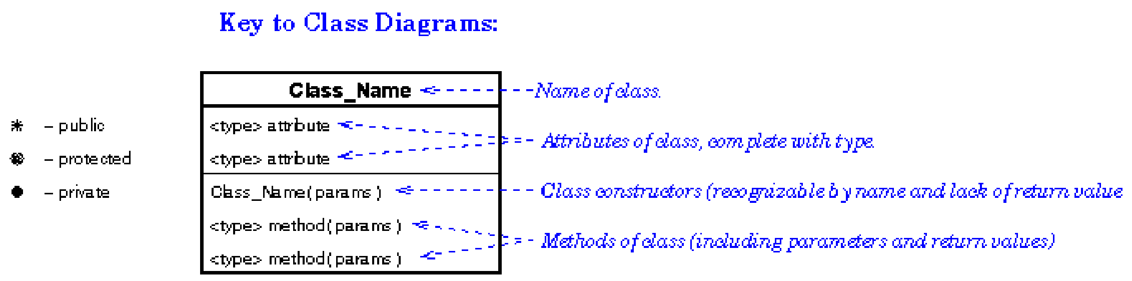


Figure 7.2: Key to class diagrams. We assume that all attributes of classes are private, and all constructors and methods are public, unless otherwise indicated.

The *Artifact* module contains the software for manipulating artifacts, such as files, lines, or "logical" Items. This version of AnnoSpec supports only certain well-defined type of Items (called *Units of Annotation* or *UAs* in the code). In this version of AnnoSpec, the system itself does not parse the artifact into Items. The Moderator of an inspection must identify the Items manually, and insert the identities of the Items into ASCII files that the AnnoSpec system reads during initialization. In large-scale use, parsing would be very worthwhile.

The logical Items supported in the present version of AnnoSpec are:

- Class – An OO class. This logical Item would also work for any non-OO aggregation of data, such as a C-language structure.
- Attribute – An attribute of an OO class
- Method – A method (function) of an OO class, or possibly a function not in any class (available in C++). This would also serve for the functions and/or subroutines of imperative languages like Fortran, C, Pascal, etc.
- Comment – A non-executable comment within source code
- Paragraph – A paragraph of text, possibly a paragraph within a comment.
- File – An entire file.

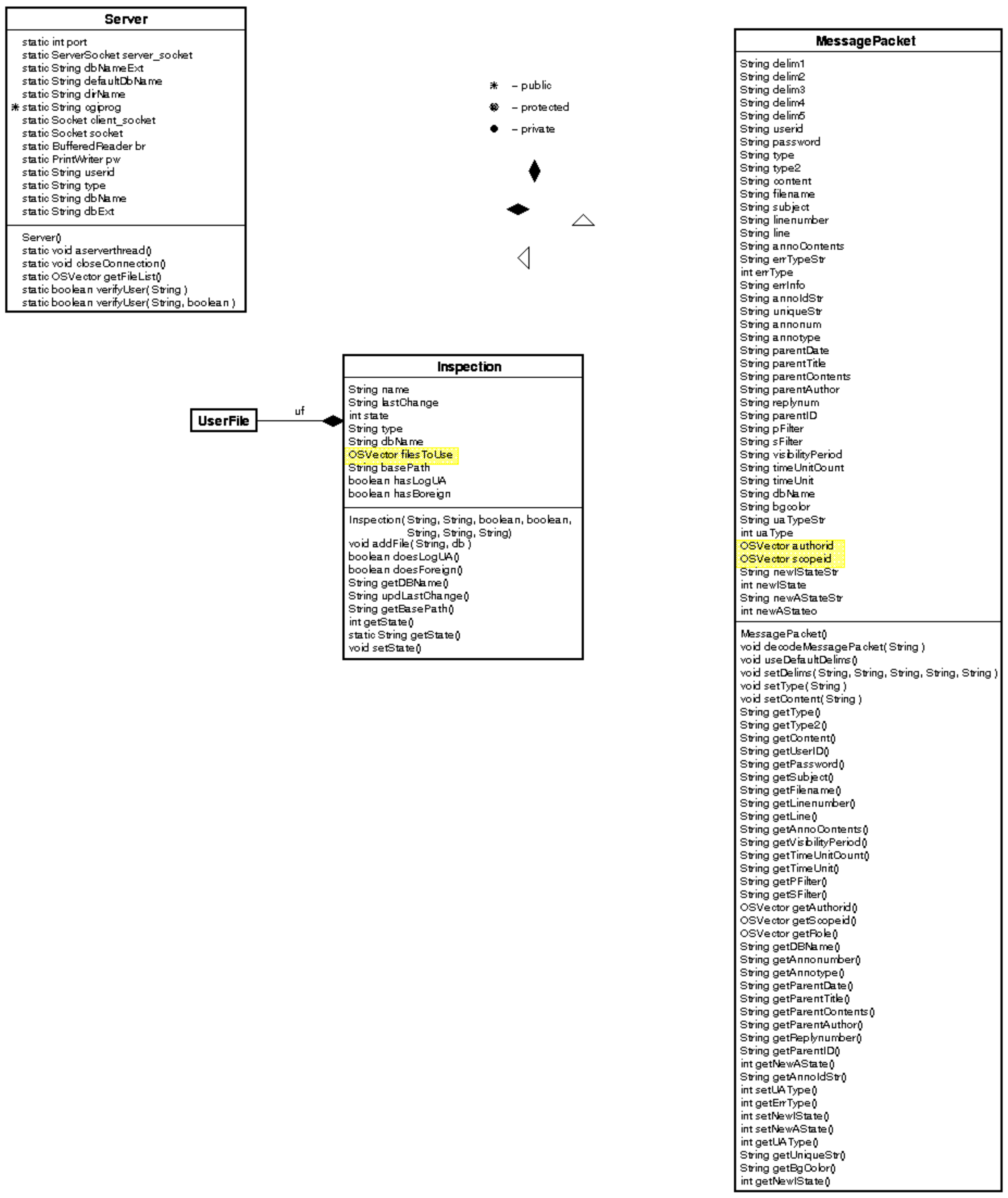


Figure 7.3: Design of the AnnoServer module.

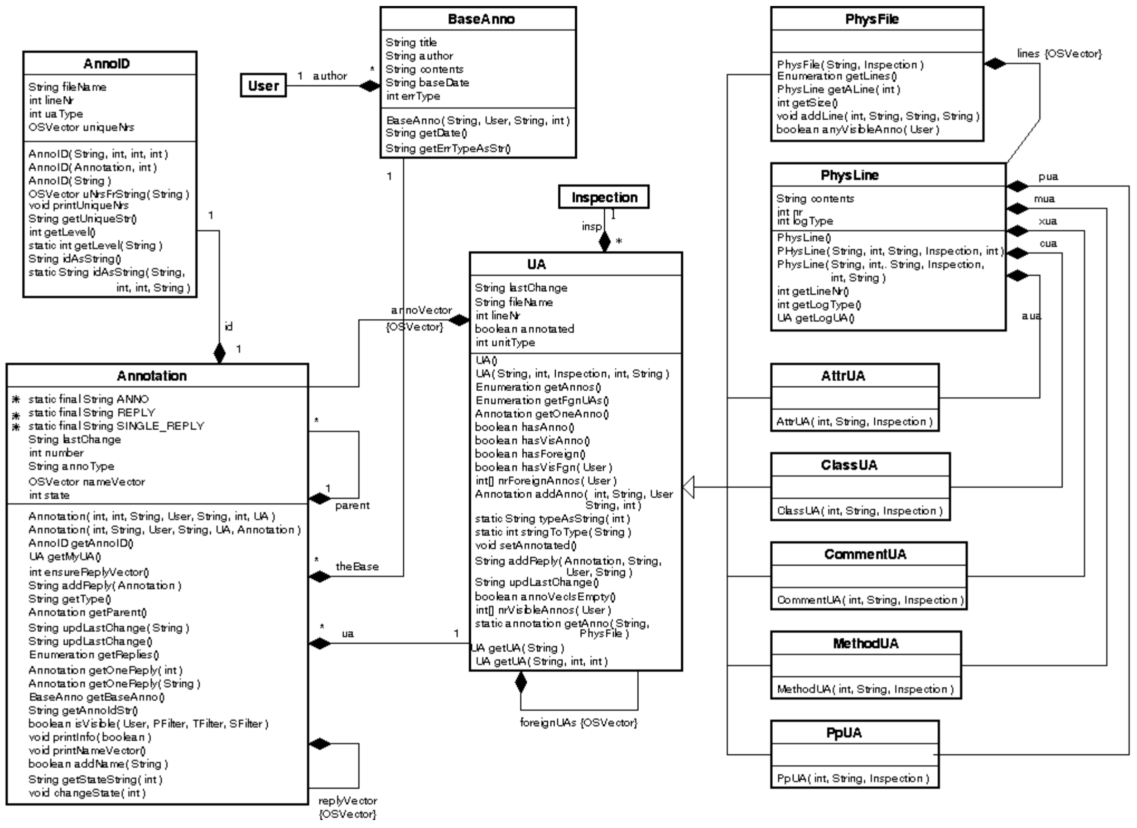

BaseAnno
String title
String author
String contents
String baseDate
int errType
BaseAnno(String, User, String, int)
String getDate()
String getErrTypeAsStr()
PhysFile
PhysFile(String, Inspection)
Enumeration getLines()
PhysLine getALine(int)
int getSize()
void addLine(int, String, String, String)
boolean anyVisibleAnno(User)
lines {OSVector}
AnnoID
String fileName
int lineNr
int uaType
OSVector uniqueNrs
AnnoID(String, int, int, int)
AnnoID(Annotation, int)
AnnoID(String)
OSVector uNrsFrString(String)
void printUniqueNrs
String getUniqueStr()
int getLevel()
static int getLevel(String)
String idAsString()
static String idAsString(String, int, int, String)
User
author
PhysLine
String contents
int nr
int logType
PhysLine()
PHysLine(String, int, String, Inspection, int)
PhysLine(String, int,. String, Inspection, int, String)
int getLineNr()
int getLogType()
UA getLogUA()
pua
mua
xua
cua
aua
Inspection
insp
UA
String lastChange
String fileName
int lineNr
boolean annotated
int unitType
UA()
UA(String, int, Inspection, int, String)
Enumeration getAnnos()
Enumeration getFgnUAs()
Annotation getOneAnno()
boolean hasAnno()
boolean hasVisAnno()
boolean hasForeign()
boolean hasVisFgn(User)
int[] nrForeignAnnos(User)
Annotation addAnno(int, String, User String, int)
static String typeAsString(int)
static int stringToType(String)
void setAnnotated()
String addReply(Annotation, String, User, String)
String updLastChange()
boolean annoVecIsEmpty()
int[] nrVisibleAnnos(User)
static annotation getAnno(String, PhysFile)
UA getUA(String)
UA getUA(String, int, int)
foreignUAs {OSVector}
annoVector {OSVector}
id
Annotation
static final String ANNO
static final String REPLY
static final String SINGLE_REPLY
String lastChange
int number
String annoType
OSVector nameVector
int state
Annotation(int, int, String, User, String, int, UA)
Annotation(int, String, User, String, UA, Annotation)
AnnoID getAnnoID()
UA getMyUA()
int ensureReplyVector()
String addReply(Annotation)
String getType()
Annotation getParent()
String updLastChange(String)
String updLastChange()
Enumeration getReplies()
Annotation getOneReply(int)
Annotation getOneReply(String)
BaseAnno getBaseAnno()
String getAnnoIdStr()
boolean isVisible(User, PFilter, TFilter, SFilter)
void printInfo(boolean)
void printNameVector()
boolean addName(String)
String getStateString(int)
void changeState(int)
parent
theBase
ua
replyVector {OSVector}
AttrUA
AttrUA(int, String, Inspection)
ClassUA
ClassUA(int, String, Inspection)
CommentUA
CommentUA(int, String, Inspection)
MethodUA
MethodUA(int, String, Inspection)
PpUA
PpUA(int, String, Inspection)

Figure 7.4: Design of the Artifact module.

The *Util* module contains utility classes that are of general usefulness. It includes many interfaces (not shown) that abstract out the parameters from the classes in the other modules.

The *UI* module contains the user interface of AnnoSpec. Each class that displays a different screen is a subclass of the general Display class. With this user interface, the user presses buttons to view and annotate artifacts. We could have used hyperlinks for navigational purposes, but we felt that buttons made for a cleaner design. However, buttons turned out to paint very slowly to browser screens.

The *Users* module contains the information used by AnnoSpec to manipulate users. This module includes the software needed for annotation filtering. There were three types of filters defined in this version of AnnoSpec.

1) **People filters** allow users to specify some subset of annotators whose annotations they wished to see. They can specify people by name, or by role (e.g., "all Reviewers").

2) **Time filters** allow people to see annotations within a certain time frame. These are limited to letting people see "recent" annotations, where "recent" can be user-defined to mean within some specified number of days, hours, or weeks of the present time, or since the user's last login to the inspection.

3) **Scope filters** allow people to see annotations of given scopes, among those listed in the above bullet list (e.g., class, file, etc.)

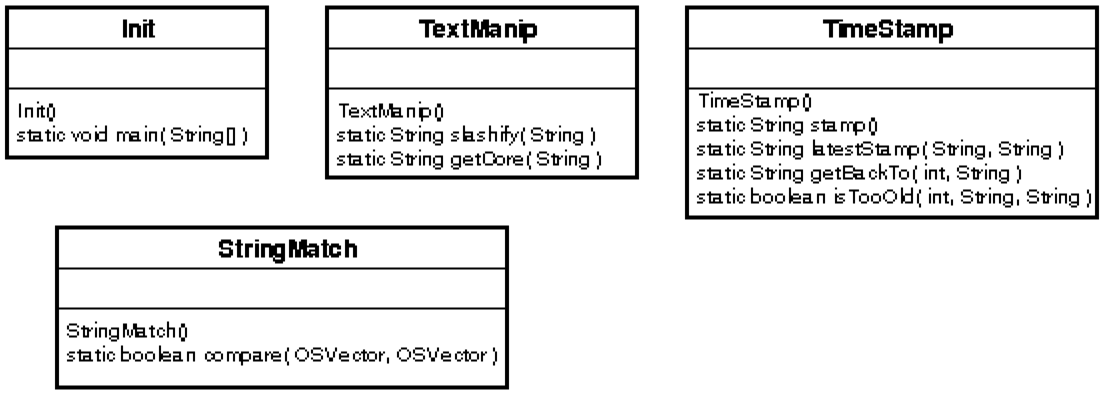

Init
Init()
static void main(String[])
TextManip
TextManip()
static String slashify(String)
static String getCore(String)
TimeStamp
TimeStamp()
static String stamp()
static String latestStamp(String, String)
static String getBackTo(int, String)
static boolean isTooOld(int, String, String)
StringMatch
StringMatch()
static boolean compare(OSVector, OSVector)

Figure 7.5: Design of the Util module. Not shown are interfaces that define global constants.

7.4 AnnoSpec implementation

The implementation differs from the design because of some idiosyncrasies with the PSE database. PSE can only store native Java types (int, char, boolean, double, float) and Java Strings as persistent objects. To get around this, we stored our persistent objects of other types as elements within an "OSVector". When we extract the OSVector from the database, we then must cast the extracted object to the correct type. PSE can also store "Enumerations" of any type. Again, we cast the enumerated items to the appropriate types before use.

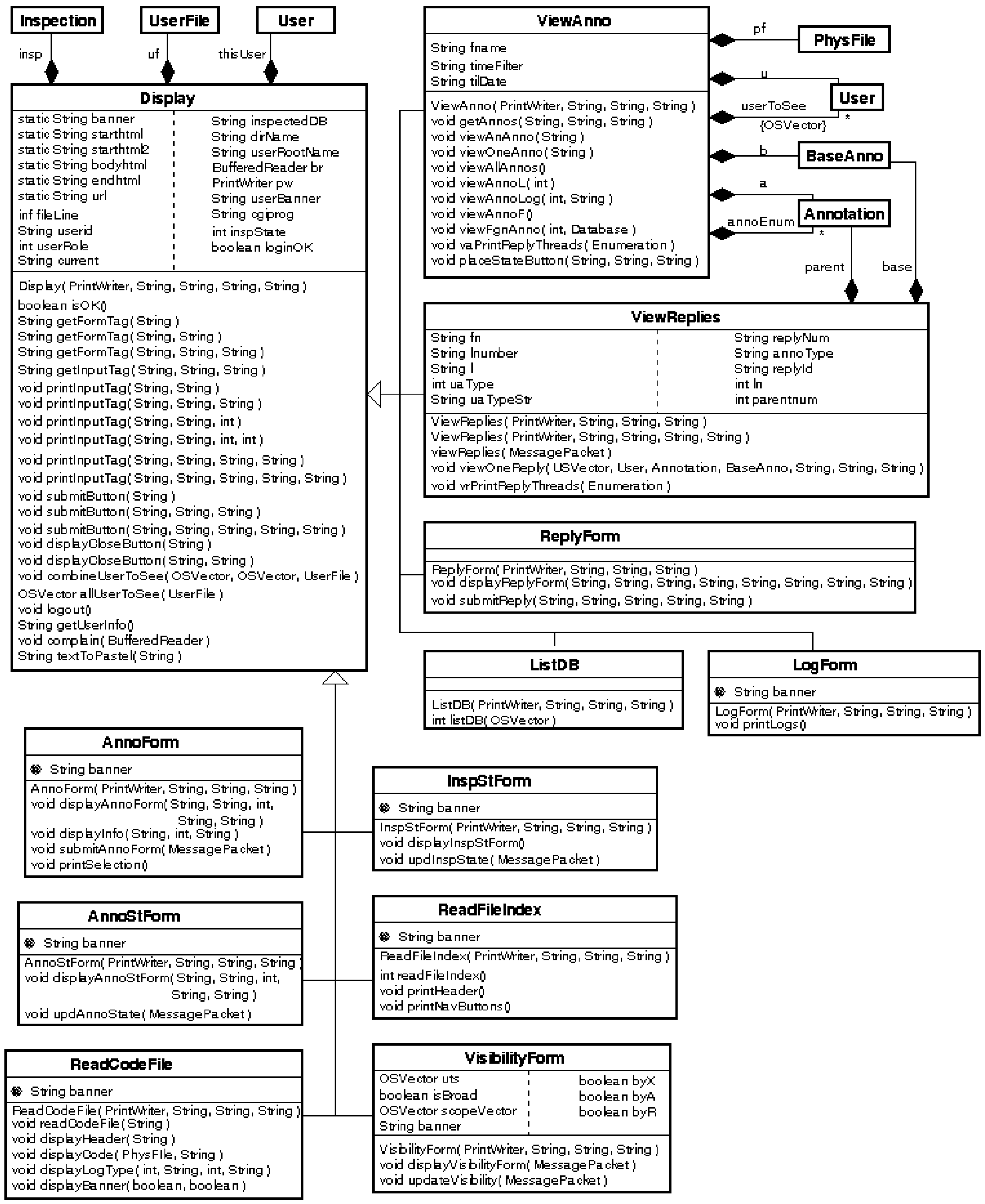

Inspection
insp
UserFile
uf
User
thisUser
Display
static String banner
static String starthtml
static String starthtml2
static String bodyhtml
static String endhtml
static String url
int fileLine
String userid
int userRole
String current
String inspectedDB
String dirName
String userRootName
BufferedReader br
PrintWriter pw
String userBanner
String cgiprog
int inspState
boolean loginOK
Display(PrintWriter, String, String, String, String)
boolean isOK()
String getFormTag(String)
String getFormTag(String, String)
String getFormTag(String, String, String)
String getInputTag(String, String, String)
void printInputTag(String, String)
void printInputTag(String, String, String)
void printInputTag(String, String, int)
void printInputTag(String, String, int, int)
void printInputTag(String, String, String, String)
void printInputTag(String, String, String, String, String)
void submitButton(String)
void submitButton(String, String, String)
void submitButton(String, String, String, String, String)
void displayCloseButton(String)
void displayCloseButton(String, String)
void combineUserToSee(OSVector, OSVector, UserFile)
OSVector allUserToSee(UserFile)
void logout()
String getUserInfo()
void complain(BufferedReader)
String textToPastel(String)
ViewAnno
String fname
String timeFilter
String tilDate
ViewAnno(PrintWriter, String, String, String)
void getAnnos(String, String, String)
void viewAnAnno(String)
void viewOneAnno(String)
void viewAllAnnos()
void viewAnnoL(int)
void viewAnnoLog(int, String)
void viewAnnoF()
void viewFgnAnno(int, Database)
void vaPrintReplyThreads(Enumeration)
void placeStateButton(String, String, String)
pf
PhysFile
u
User
userToSee
{OSVector}
*
b
BaseAnno
a
Annotation
annoEnum
*
parent
base
ViewReplies
String fn
String lnumber
String l
int uaType
String uaTypeStr
String replyNum
String annoType
String replyId
int ln
int parentnum
ViewReplies(PrintWriter, String, String, String)
ViewReplies(PrintWriter, String, String, String, String)
viewReplies(MessagePacket)
void viewOneReply(USVector, User, Annotation, BaseAnno, String, String, String)
void vrPrintReplyThreads(Enumeration)
ReplyForm
ReplyForm(PrintWriter, String, String, String)
void displayReplyForm(String, String, String, String, String, String, String)
void submitReply(String, String, String, String, String)
ListDB
ListDB(PrintWriter, String, String, String)
int listDB(OSVector)
LogForm
String banner
LogForm(PrintWriter, String, String, String)
void printLogs()
AnnoForm
String banner
AnnoForm(PrintWriter, String, String, String)
void displayAnnoForm(String, String, int, String, String)
void displayInfo(String, int, String)
void submitAnnoForm(MessagePacket)
void printSelection()
InspStForm
String banner
InspStForm(PrintWriter, String, String, String)
void displayInspStForm()
void updInspState(MessagePacket)
AnnoStForm
String banner
AnnoStForm(PrintWriter, String, String, String)
void displayAnnoStForm(String, String, int, String, String)
void updAnnoState(MessagePacket)
ReadFileIndex
String banner
ReadFileIndex(PrintWriter, String, String, String)
int readFileIndex()
void printHeader()
void printNavButtons()
ReadCodeFile
String banner
ReadCodeFile(PrintWriter, String, String, String)
void readCodeFile(String)
void displayHeader(String)
void displayCode(PhysFile, String)
void displayLogType(int, String, int, String)
void displayBanner(boolean, boolean)
VisibilityForm
OSVector uts
boolean isBroad
OSVector scopeVector
String banner
boolean byX
boolean byA
boolean byR
VisibilityForm(PrintWriter, String, String, String)
void displayVisibilityForm(MessagePacket)
void updateVisibility(MessagePacket)

Figure 7.6: Design of the User Interface (UI) module.

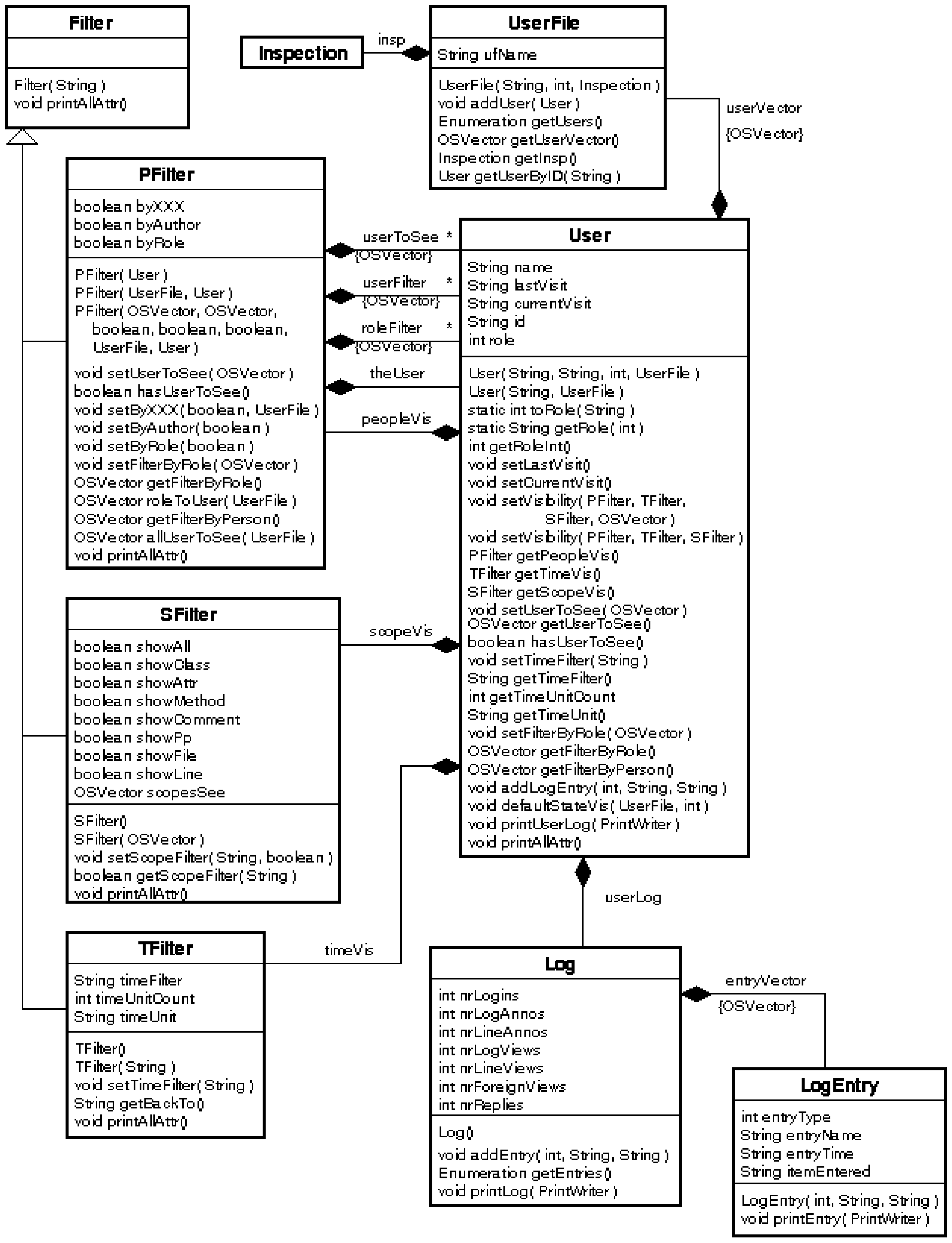


Figure 7.7: Design of the Users module.

7.5 Use of AnnoSpec

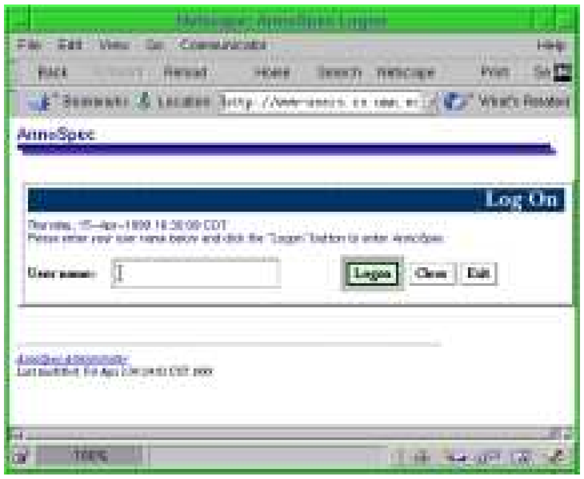
This section describes how AnnoSpec works in practice, illustrated with a set of screens showing how a user could annotate a very short C++ method, read the annotations of others, and apply filters to address annotation visibility. The screen shots shown were taken from an AnnoSpec annotation session run using the Netscape browser under the Linux operating system. AnnoSpec looks slightly different using the Internet Explorer browser, and also looks slightly different between Linux and Windows versions of either browser.

We describe the use of AnnoSpec in the context of a software inspection conducted according to the protocol described in Chapter 4. Using AnnoSpec for an informal annotation event would be a subset of the work described in this section. The particular subset would depend on the degree of formality desired. AnnoSpec supports formal software inspection containing the Fault Collection and Discussion states. We also support an "Anno Only" state for informal annotation events, and an archival "Done" state. Formal inspection also gives participants defined roles. We indicate when screens only make sense for people in certain roles and/or for an inspection in certain states. For other screens, we note when the use of AnnoSpec is dependent on the phase of inspection, and when the use of AnnoSpec is role-specific.

We discuss the following activities using AnnoSpec: Logging in, selecting a database or inspection, selecting a file to view, manipulating the chosen View, annotating a file, viewing an annotation, replying to an annotation, and changing the state of an annotation or an inspection.

7.5.1 Logging In

AnnoSpec is designed for logging in from a URL. Once a user accesses the URL, that user can type in their login name and press the login button, as shown in Figure 7.8. In the present version of AnnoSpec, a password is not required.



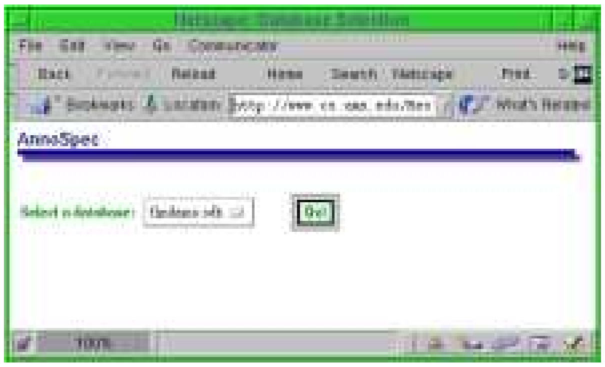


Figure 7.8: AnnoSpec login screen.

Figure 7.9: Database selection screen.

7.5.2 Database Selection

The Database Selection screen (Figure 7.9) lets a user choose the database for the inspection. The pull-down menu allows users to choose among all inspections in which they are participants.

7.5.3 File Selection

An inspection may be divided up into multiple files. Using the File Index screen (Figure 7.10), the user can choose which file to view. For each file, AnnoSpec indicates whether the file has been updated since the user's last login, and whether there are any annotations visible to the user on that file.

In the File Index screen, a user can press one of the numbered buttons on the left to select a file to inspect. The buttons along the top of the File Index screen let users set annotation visibility, print logs of activities, or change inspection state (inspection state and user role permitting).

7.5.3.1 State and Role Dependencies

In the Fault Collection phase of inspection, Reviewers can only see their own annotations, to prevent "groupthink" and "free-riding" on the effort of others. However, they can still filter annotations by time or scope.

Since the Moderator's duties in an inspection include making sure that progress is being made, the Moderator must be able to view the work of other people. Other members of a formal inspection team cannot see any annotations but their own. Similarly, only the Moderator can print user logs to see at a glance what work the members of the inspection team have done. Finally, only the Moderator can change the state of the inspection. Thus, the “Change Inspection State” and “Print User Logs” buttons do not appear on this screen for users in any other role.

7.5.4 Viewing a File to Inspect

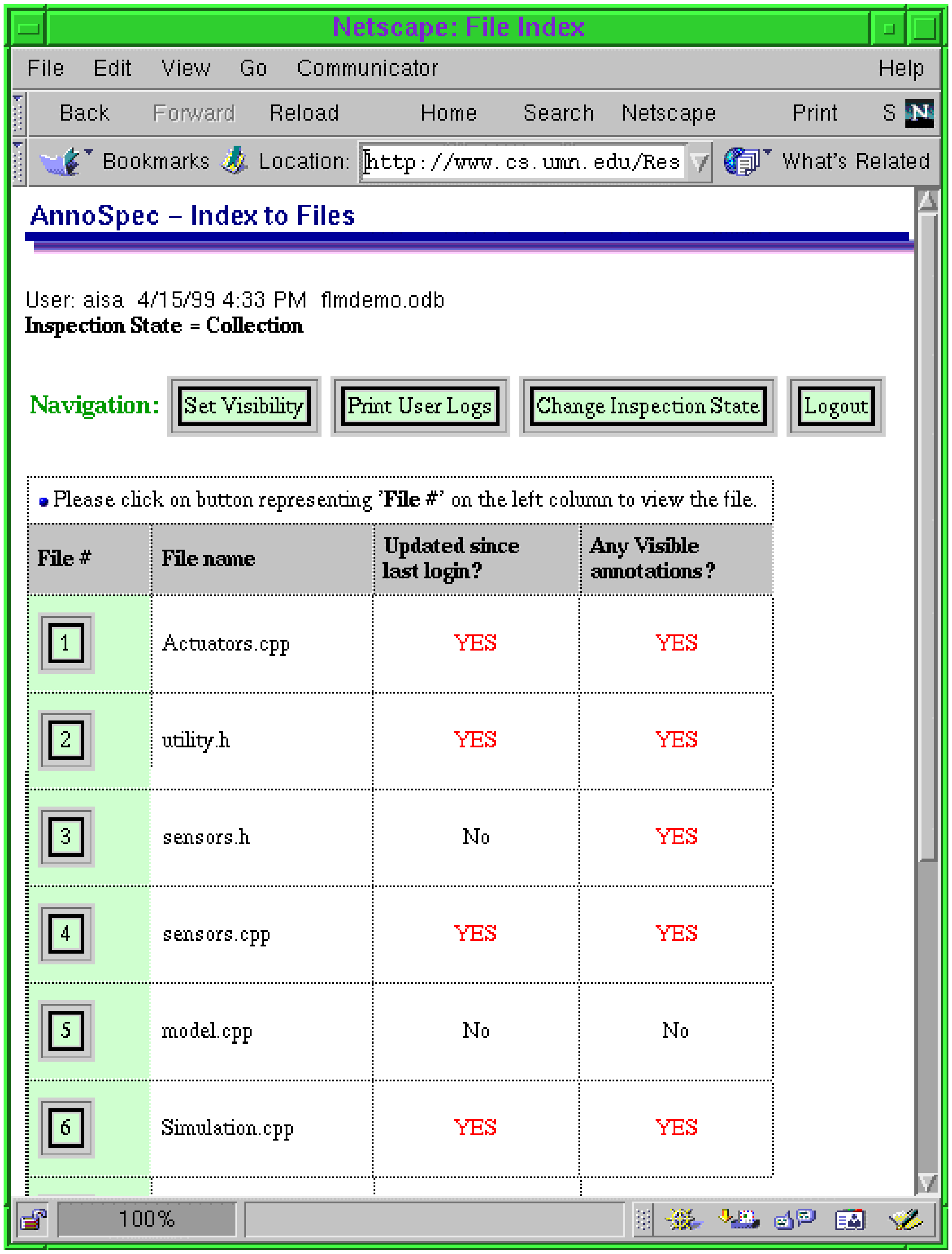
When a user selects a file, a screen akin to Figure 7.11 appears on their main browser window. Users can use the green navigation buttons at the top to go back to the file index, to set annotation visibility when permitted by the inspection protocol, or to log out.

As described previously, annotations differ in their scopes. Some annotations have a very narrow scope, such as a part of a single line. An example would be a misspelling or the use of the wrong number in a constant. These can be said to have a "line" scope for our purposes, because a line is the smallest unit the AnnoSpec tool allows us to annotate. But some annotations have broader scopes. For instance, if someone thinks that a given method (or function) is trying to do the wrong thing, the scope of that annotation would be the entire method. Scopes supported by AnnoSpec at present are:

- Line
- Entire File
- Class

- Attribute
- Method
- Comment (for multiline comments)

To annotate a line, as user pushes a cyan button in the "Line Annotate" column. To annotate an entire file, a user presses the magenta "Annotate " button. To annotate Items of Class, Attribute, Method, or Comment scopes a user presses the appropriate blue button on the far left.


Netscape: File Index
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print S
Bookmarks Location: http://www.cs.umn.edu/Res What's Related
100%

AnnoSpec – Index to Files

User: aisa 4/15/99 4:33 PM flmdemo.odb
Inspection State = Collection

Navigation: Set Visibility | Print User Logs | Change Inspection State | Logout

• Please click on button representing '**File #**' on the left column to view the file.

File #	File name	Updated since last login?	Any Visible annotations?
1	Actuators.cpp	YES	YES
2	utility.h	YES	YES
3	sensors.h	No	YES
4	sensors.cpp	YES	YES
5	model.cpp	No	No
6	Simulation.cpp	YES	YES

Figure 7.10: File selection screen.

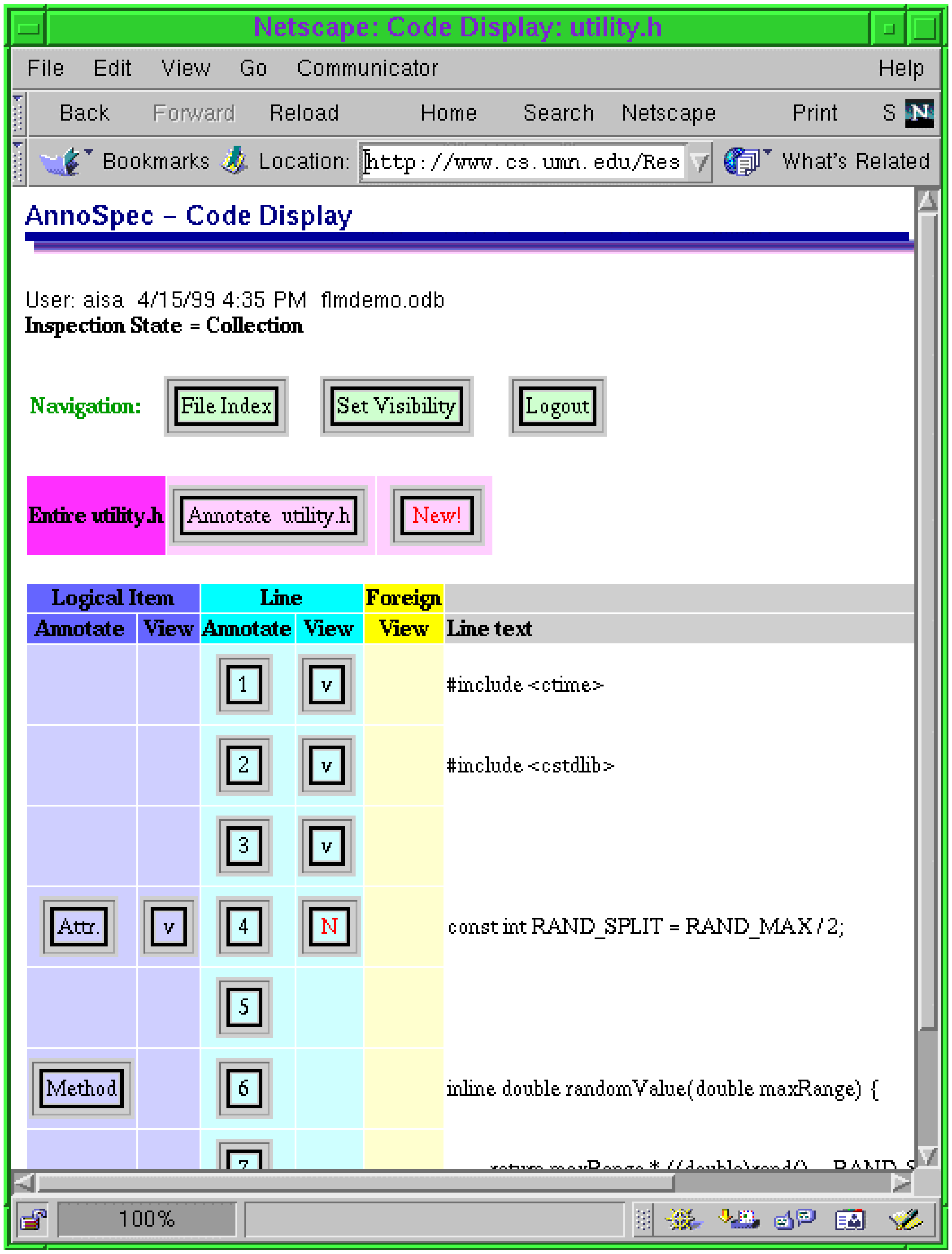
The annotation columns do not appear after the inspection is in the Done state. Buttons only appear in a View column if there exists an annotation that the viewer, in a given role, is permitted to view in that state. For instance, a view button will not exist for a Reviewer in the Fault Collection state unless that Reviewer has annotated that specific line or Item.

Any given line or logical Item may have associated with it multiple annotations, only some of which are visible to the user. The existence of a button in a given row of a “view” column means that there is at least one visible annotation in that location. Different users may press the same “view” button and see different subsets of all the annotations on that line or logical Item because of their differing roles in the inspection, and/or because of differences in the way they set their visibility filters (see Section 7.5.7).

7.5.5 Adding an Annotation

Figure 7.12 illustrates a filled-out form for annotating a line. The background color is keyed to the color of the button pressed. This form appears in the "Form Window" of the tool, a new window that AnnoSpec opens when an annotation is to be added. The artifact remains visible in the main window. The user must enter three pieces of information before submitting an annotation by pressing the green “Submit” button.

1. The subject or title of the annotation. This should be one line.
2. The contents of the annotation. This may be entered in as much detail as desired
3. The type of the annotation, chosen from one of the buttons on the lower part of the form.


Netscape: Code Display: utility.h
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print S
Bookmarks Location: http://www.cs.umn.edu/Res What's Related
100%

AnnoSpec – Code Display

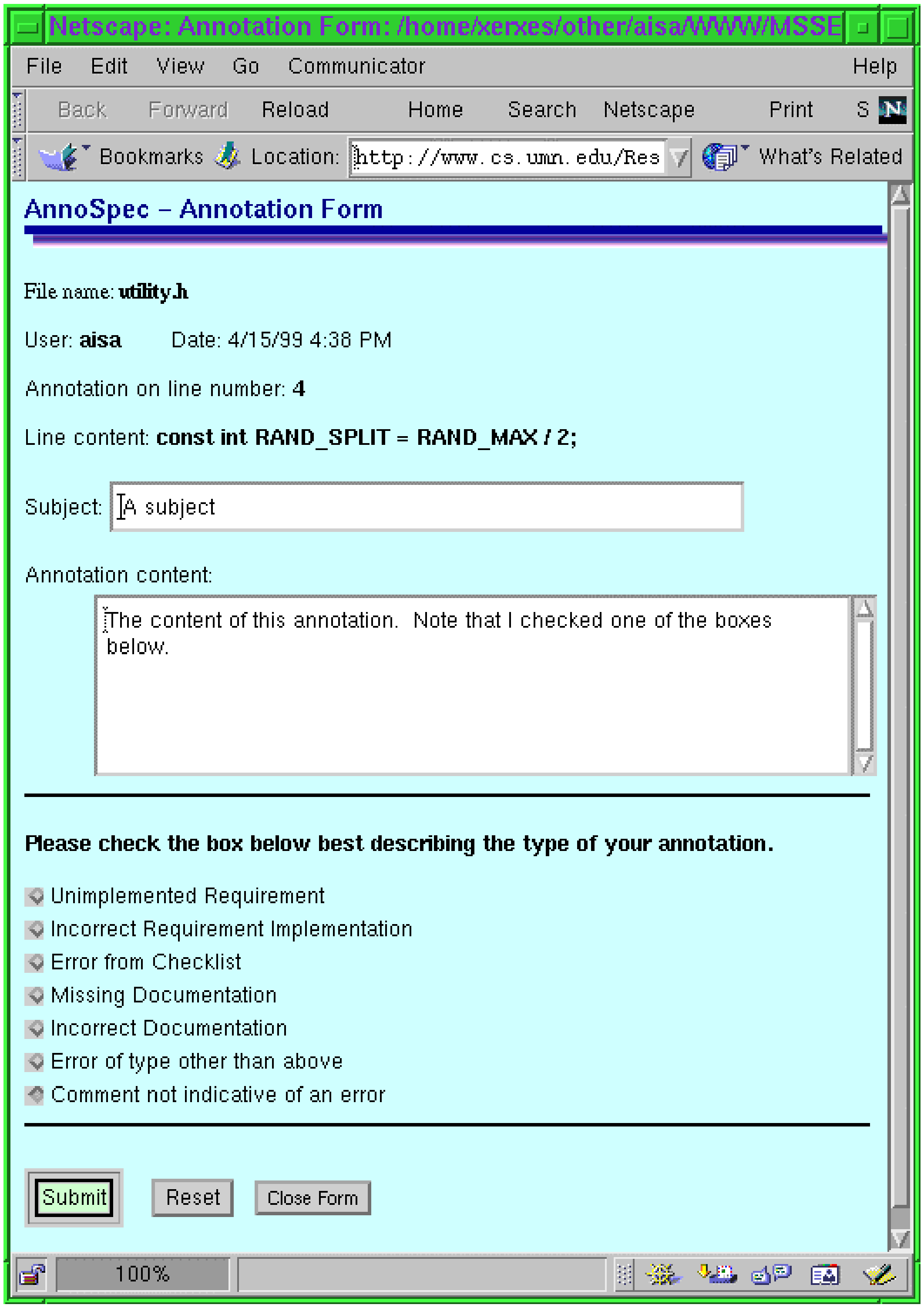
User: aisa 4/15/99 4:35 PM flmdemo.odb
Inspection State = Collection

Navigation: File Index | Set Visibility | Logout

Entire utility.h | Annotate utility.h | New!

Logical Item		Line		Foreign	
Annotate	**View**	**Annotate**	**View**	**View**	**Line text**
		1	v		#include <ctime>
		2	v		#include <cstdlib>
		3	v		
Attr.	v	4	N		const int RAND_SPLIT = RAND_MAX / 2;
		5			
Method		6			inline double randomValue(double maxRange) {
		7			return maxRange * ((double)rand() / RAND_S

Figure 7.11: File Viewing Screen


Netscape: Annotation Form: /home/xerxes/other/aisa/WWW/MSSE
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print S
Bookmarks Location: http://www.cs.umn.edu/Res
What's Related
100%

AnnoSpec – Annotation Form

File name: **utility.h**

User: **aisa** Date: 4/15/99 4:38 PM

Annotation on line number: **4**

Line content: **const int RAND_SPLIT = RAND_MAX / 2;**

Subject: A subject

Annotation content:

The content of this annotation. Note that I checked one of the boxes below.

Please check the box below best describing the type of your annotation.

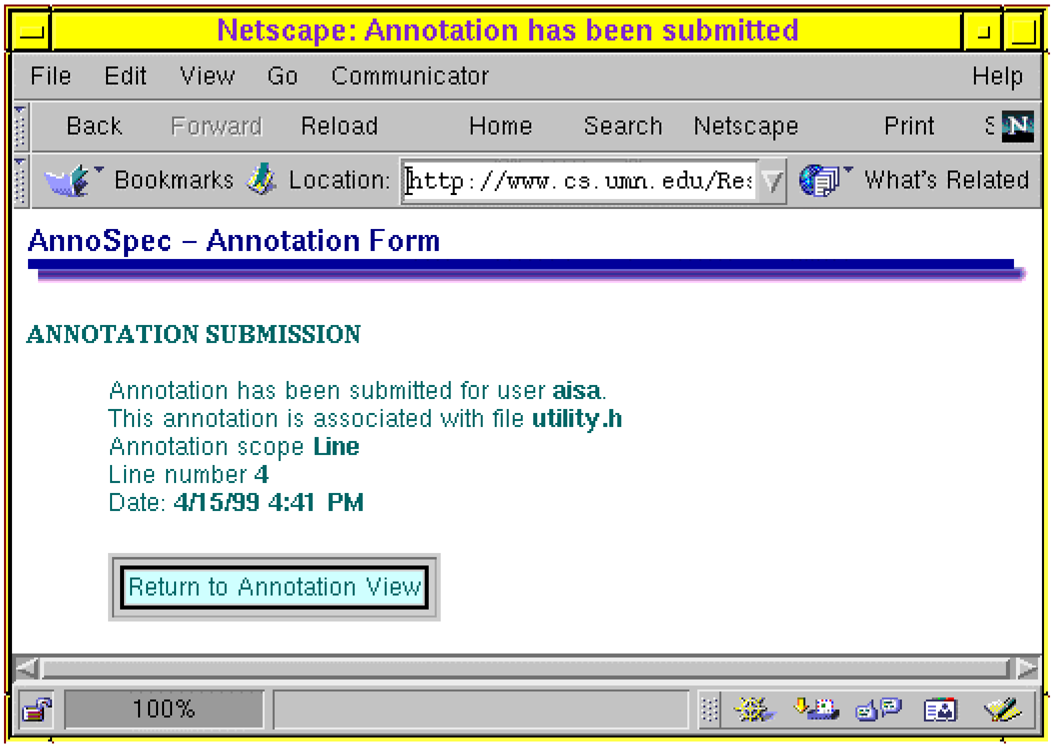
- ○ Unimplemented Requirement
- ○ Incorrect Requirement Implementation
- ○ Error from Checklist
- ○ Missing Documentation
- ○ Incorrect Documentation
- ○ Error of type other than above
- ● Comment not indicative of an error

Figure 7.12: Annotation screen.

7.5.6 Viewing an Annotation

7.5.6.1 Viewing a Submitted Annotation

After a user has submitted an annotation, Figure 7.13 appears in the Form Window, and shows that the annotation was successfully entered into that database for the inspection. The user can press the “Return to Annotation View” button to see the annotation.


Netscape: Annotation has been submitted
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Location: http://www.cs.umn.edu/Re
What's Related
100%

AnnoSpec – Annotation Form

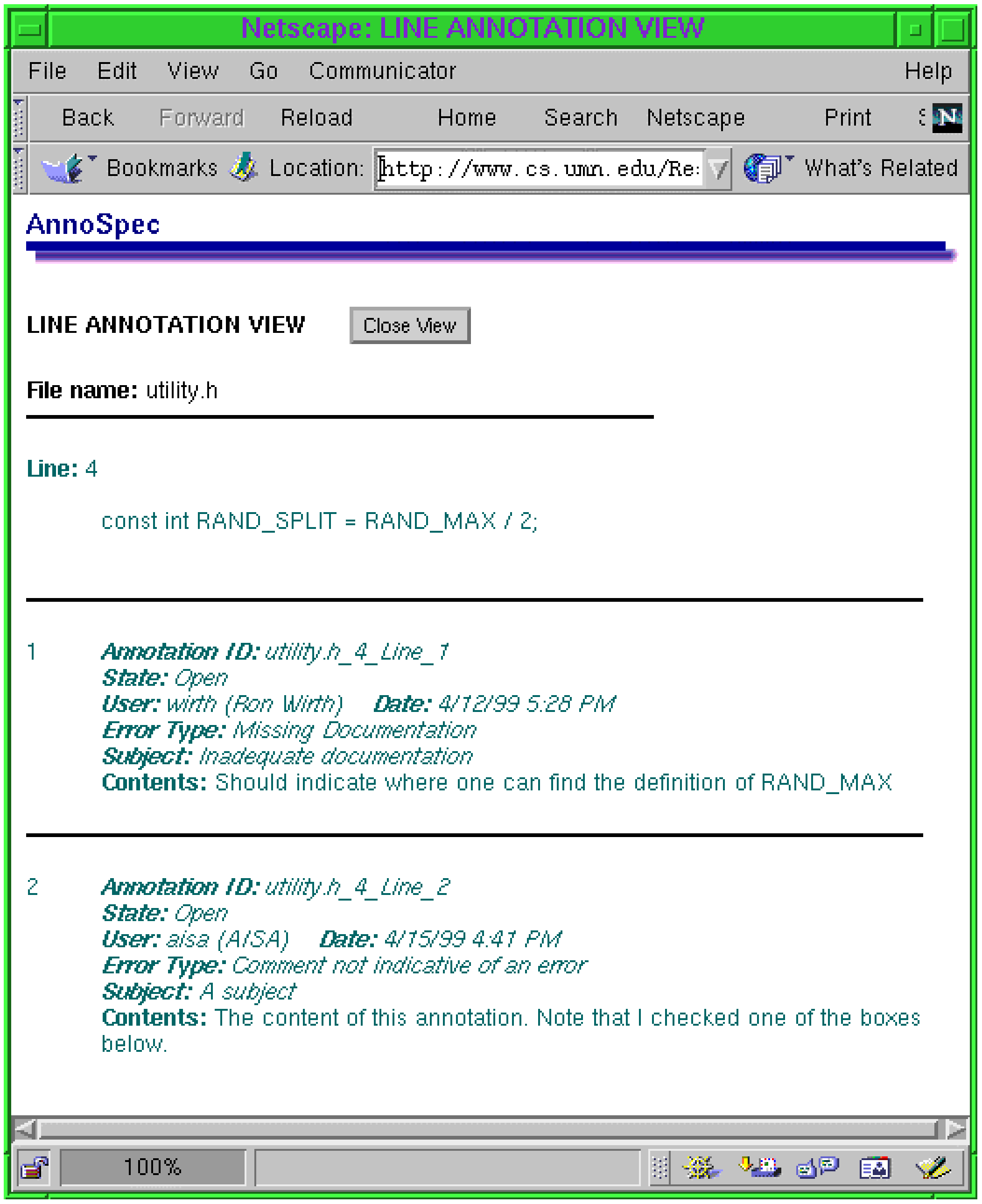
ANNOTATION SUBMISSION

Annotation has been submitted for user **aisa**.
This annotation is associated with file **utility.h**
Annotation scope **Line**
Line number **4**
Date: **4/15/99 4:41 PM**

Return to Annotation View

Figure 7.13: Annotation submission confirmation

The annotation appears as in Figure 7.14, along with any other annotations made to the same Item, if those annotations are visible to the user (that is, if they pass that user’s filter). Such a view will show only annotations of the given scope. For instance, on line 4 of the artifact in Figure 7.11, pressing the View Logical Item button in line 4 will show all annotations made on logical Items, and pressing the View Line button in line 4 will show all annotations made with line scope on line 4.


Netscape: LINE ANNOTATION VIEW
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Location: http://www.cs.umn.edu/Re What's Related
100%

AnnoSpec

LINE ANNOTATION VIEW Close View

File name: utility.h

Line: 4

const int RAND_SPLIT = RAND_MAX / 2;

1 ***Annotation ID:*** *utility.h_4_Line_1*
State: *Open*
User: *wirth (Ron Wirth)* ***Date:*** *4/12/99 5:28 PM*
Error Type: *Missing Documentation*
Subject: *Inadequate documentation*
Contents: Should indicate where one can find the definition of RAND_MAX

2 ***Annotation ID:*** *utility.h_4_Line_2*
State: *Open*
User: *aisa (AISA)* ***Date:*** *4/15/99 4:41 PM*
Error Type: *Comment not indicative of an error*
Subject: *A subject*
Contents: The content of this annotation. Note that I checked one of the boxes below.

Figure 7.14: View of annotations of an Item.

7.5.6.2 Seeing an Annotation from the Code

Figure 7.11 is the screen from which a user browses the code in the main window. Pressing a button under one of the "View" columns shows the user all the annotations on that line or item (e.g., Method, Attribute), in the format of Figure 7.14.

The button will usually be marked with a "v" for "view", but sometimes it will have a red "N" for "New". New annotations are those which have been added since the last time the user visited the page, whether this was in a previous login session or earlier in the user’s present login session. Marking new annotations is important in this screen, because it shows the users where the new annotations are. The file index screen only shows which files have new annotations, not where those new annotations are within the files.

7.5.6.3 Foreign Annotations

The yellow column with view buttons allows a user to view foreign annotations made on another part of the document. Foreign annotations might be of interest to a viewer not looking at the part of the document on which the annotation was made, as described in Chapter 3.

For instance, an annotation of the signature of the method in a C++ header (“.h”) file, saying that the parameters or return type should be changed, is potentially interesting to someone looking at the C++ source code (".cpp") file that implements that method. So the AnnoSpec tool would show a foreign annotation in the source code file on the line containing the method declaration.

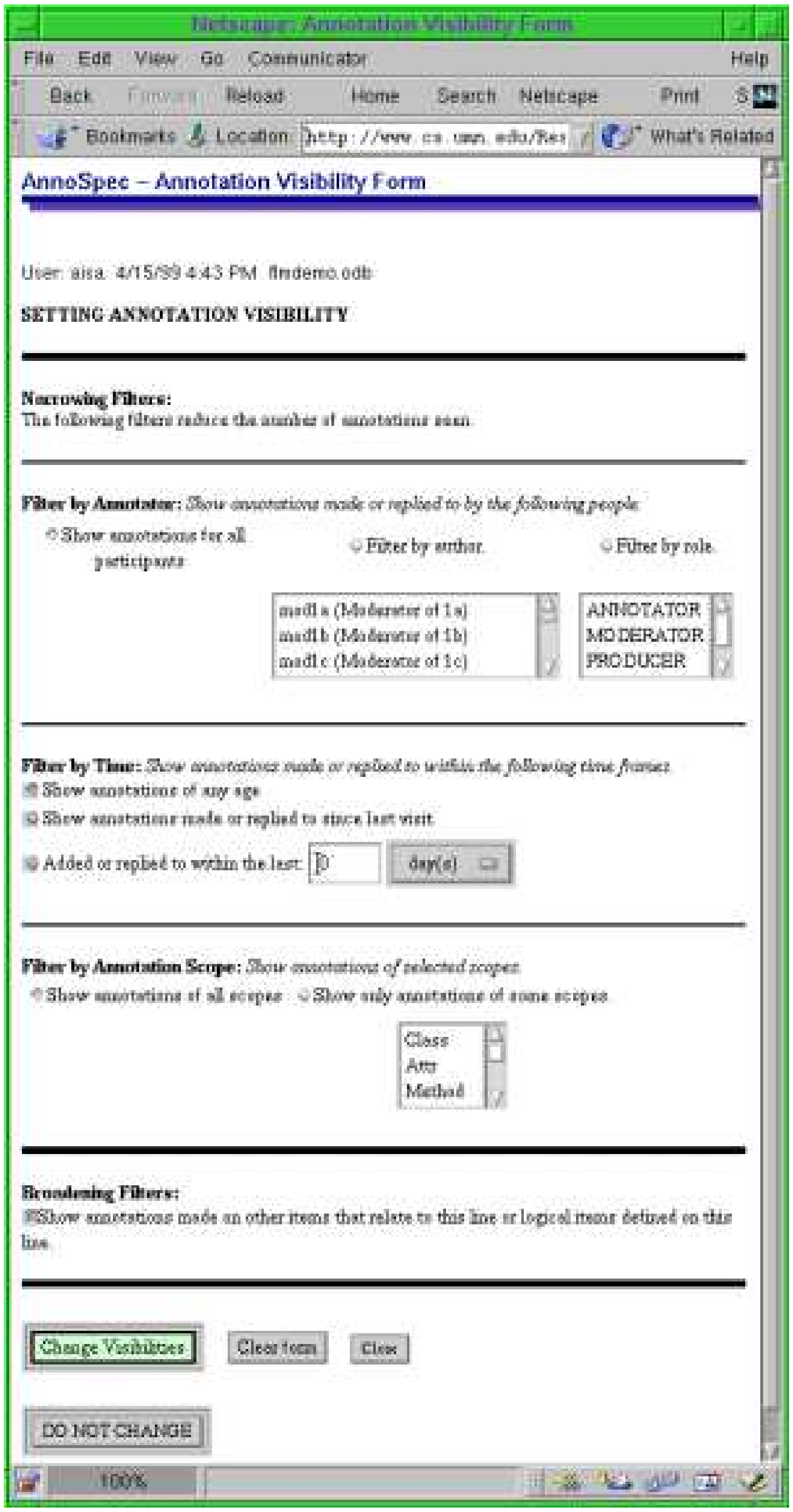
7.5.6.4 State and Role Dependencies

State and role dependencies on viewing an annotation are described in section 7.5.4.2. Note that the only ways to view an annotation are to view one’s own annotation after adding it, or to view an annotation by pressing a button in a “view” column.

7.5.7 Setting Annotation Visibility

Figure 7.15 shows the Visibility Form that the users can submit to select which annotations they want to see. The defaults are state and role dependent. They are:

1) In the Fault Collection phase, the Moderator sees all annotations, and the Reviewers see only their own annotations (they can't change this). Everybody sees annotations regardless of the time they were made and regardless of their scope, unless they choose to narrow-filter along these dimensions.


Netscape: Annotation Visibility Form
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Location: http://www.cs.umn.edu/Res What's Related

AnnoSpec – Annotation Visibility Form

User: aisa. 4/15/99 4:43 PM. flmdemo.odb

SETTING ANNOTATION VISIBILITY

Narrowing Filters:
The following filters reduce the number of annotations seen.

Filter by Annotator: *Show annotations made or replied to by the following people.*

Show annotations for all participants | Filter by author. | Filter by role.

medl a (Moderator of 1a)
medl b (Moderator of 1b)
medl c (Moderator of 1c)

ANNOTATOR
MODERATOR
PRODUCER

Filter by Time: *Show annotations made or replied to within the following time frames.*

Show annotations of any age

Show annotations made or replied to since last visit

Added or replied to within the last: 0 day(s)

Filter by Annotation Scope: *Show annotations of selected scopes.*

Show annotations of all scopes | Show only annotations of some scopes

Class
Attr
Method

Broadening Filters:
Show annotations made on other items that relate to this line or logical items defined on this line.

Figure 7.15: Visibility form

2) In the Discussion phase, by default everybody sees all annotations. However, a user can change this visibility by choosing to see annotations from only a subset of people, by choosing to see only recent annotations and specifing the timeframe that defines “recent”, or by choosing to see only annotations of certain scopes.

After the user submits a Visibility Form, the AnnoSpec tool returns the user to the File Index screen. When that user next selects a file, the visible annotations will be those annotations that have passed the filter defined on the Visibility Form, which may include foreign annotations.

The default visibility for any user in any state is the largest set of annotations that user would be permitted to see. In particular, the defaults include foreign annotations. In no case can a user change their visibility to see annotations not in their default visibility.

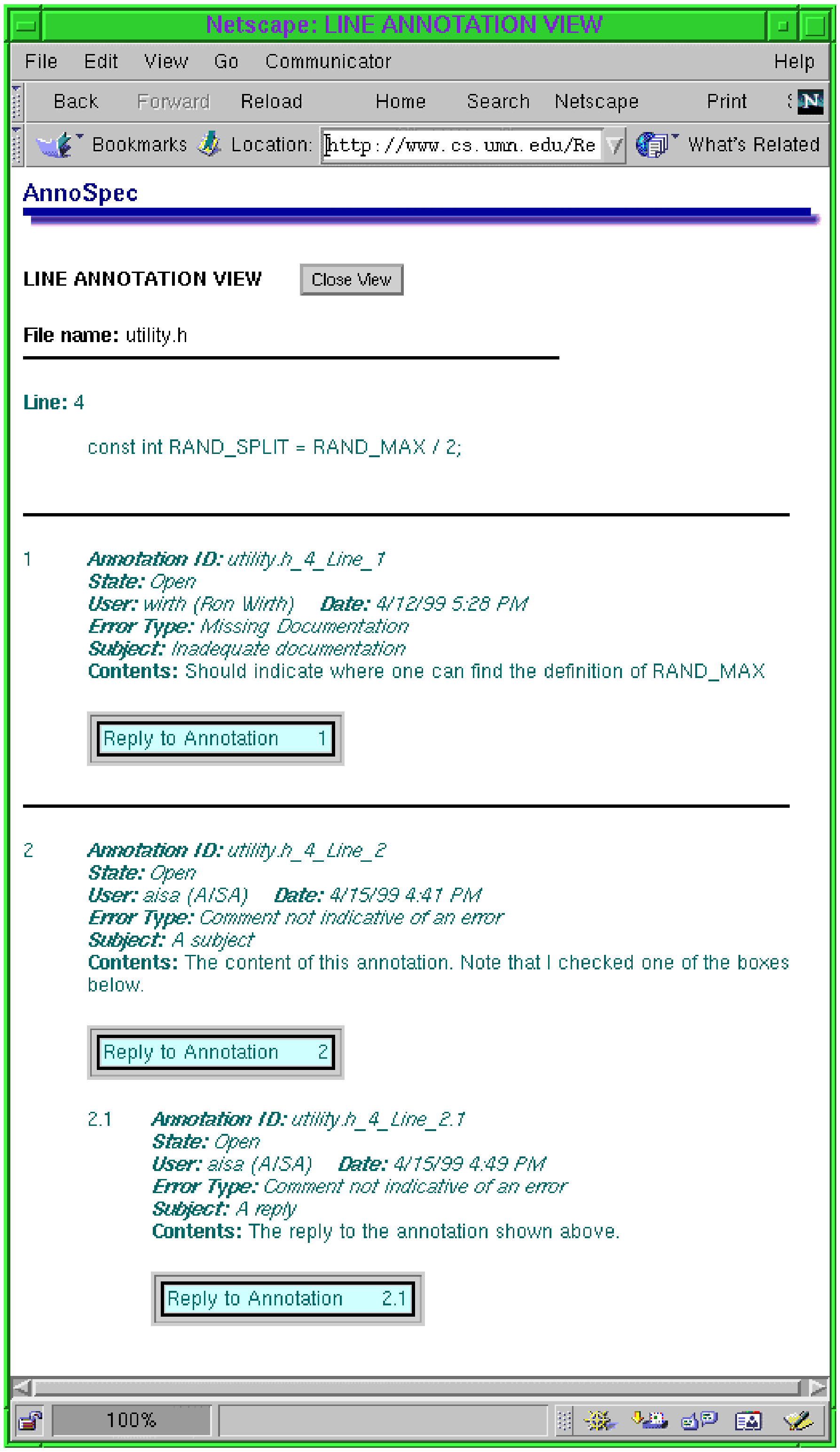
7.5.8 Discussing an Annotation

Discussing an annotation means replying to the annotation, or to someone else's reply to that annotation. Effectively, the annotation becomes the head of a Usenet-like discussion thread. The threaded discussion performs a function similar to discussing the annotation in a meeting.

Discussing an annotation is done from the Annotation view. In order to be able to reply to an annotation, the inspection must be in a state ("Anno Only" or "Discussion") that permits addition of discussion comments, and the annotation must be in the "open" state - that is the Moderator has not closed off the Annotation to future discussion.

In the Discussion phase, there is a new “reply” button added to each annotation (and reply) in the Annotation view, as shown in Figure 7.16.

To discuss the annotation, a user selects the "reply" button corresponding to the comment in the annotation discussion thread to which she wishes to reply. Then a reply form pops up. Filling out this form is much like filling out the original annotation form. However, instead of containing the text of the line being annotated, the reply form contains the text of the comment to which the reply is being directed. The user directs the reply specifically to the annotation or to a particular reply. For replies the form has a white background.


Netscape: LINE ANNOTATION VIEW
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Location: http://www.cs.umn.edu/Re What's Related

AnnoSpec

LINE ANNOTATION VIEW

File name: utility.h

Line: 4

const int RAND_SPLIT = RAND_MAX / 2;

1 ***Annotation ID:*** *utility.h_4_Line_1*
State: *Open*
User: *wirth (Ron Wirth)* ***Date:*** *4/12/99 5:28 PM*
Error Type: *Missing Documentation*
Subject: *Inadequate documentation*
Contents: Should indicate where one can find the definition of RAND_MAX

2 ***Annotation ID:*** *utility.h_4_Line_2*
State: *Open*
User: *aisa (AISA)* ***Date:*** *4/15/99 4:41 PM*
Error Type: *Comment not indicative of an error*
Subject: *A subject*
Contents: The content of this annotation. Note that I checked one of the boxes below.

2.1 ***Annotation ID:*** *utility.h_4_Line_2.1*
State: *Open*
User: *aisa (AISA)* ***Date:*** *4/15/99 4:49 PM*
Error Type: *Comment not indicative of an error*
Subject: *A reply*
Contents: The reply to the annotation shown above.

Figure 7.16: Threaded discussion of an annotation.

7.5.9 Changing Inspection State

The Moderator can change the state of the inspection by selecting the "Change Inspection State" button on the File Selection page. This brings up the “Inspection State Change Form” of Figure 7.17. The Moderator may change the inspection to one of the following states:

1. **Anno Only** The "Anno Only" state is for annotating a document outside of formal software inspection. Everyone has default visibility of all annotations, and anyone can reply to an annotation at any time.

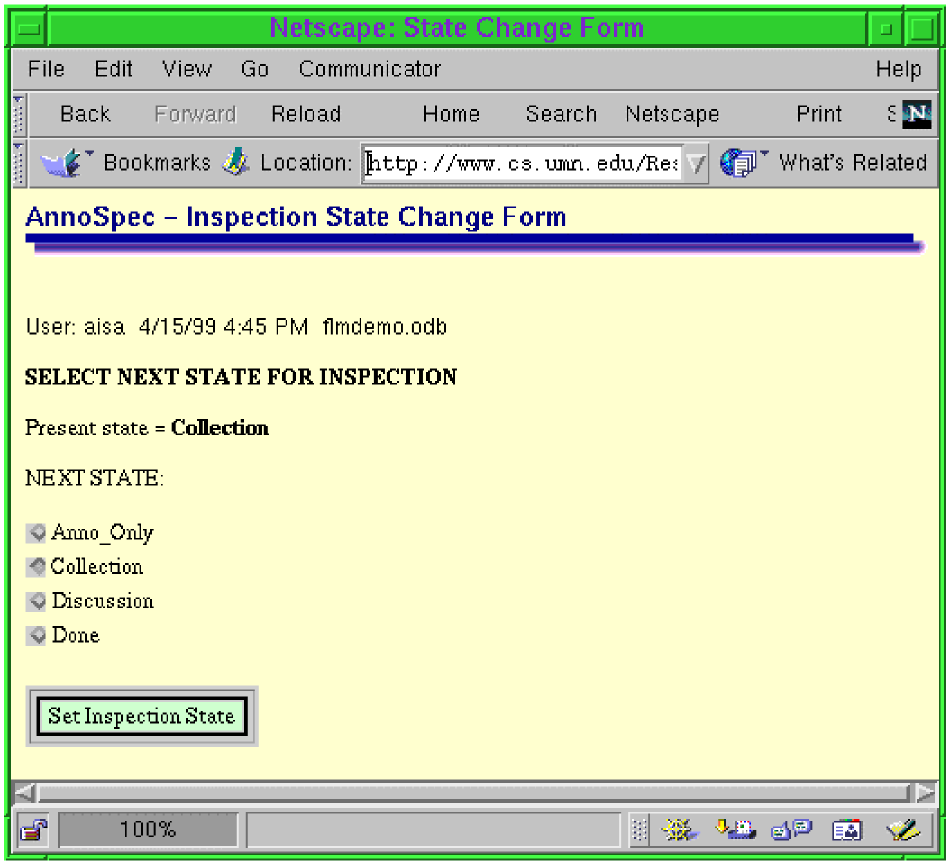
2. **Fault Collection** The Fault Collection state is for collecting faults in a formal software inspection. During this state, each inspector can see only their own annotations. This is normally the initial state of an inspection, and it is followed by the "Discussion" state.

3. **Discussion** The Discussion state is for discussing the annotations made in the Fault Collection state. During this state, each inspector sees all annotations and replies made by everyone in the inspection, can reply to annotations at will, and can make new annotations. During this state, the annotations are normally in the "open" state, but the Moderator may choose to change their state to either "Resolved" or "Unresolvable" during this time (see Changing the State of an Annotation), thus terminating discussion for this annotation.

4. **Done** The Done state is entered when the inspection is complete, and is ready to be archived. No further annotations or discussion comments are permitted.

7.5.10 Changing the State of an Annotation

The Moderator’s version of the Annotation View (Figure 7.14) contains a button to allow the Moderator to change the state of an annotation. By pressing this button, the Moderator gets an annotation State Change Form, similar to Figure 7.17 for inspection state changes. The Moderator may set the annotation state to one of the following choices:


Netscape: State Change Form
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print
Bookmarks Location: http://www.cs.umn.edu/Re What's Related
100%

AnnoSpec – Inspection State Change Form

User: aisa 4/15/99 4:45 PM flmdemo.odb

SELECT NEXT STATE FOR INSPECTION

Present state = **Collection**

NEXT STATE:

Anno_Only [✓ Collection] Discussion Done

Set Inspection State

Figure 7.17: Changing inspection state

1. **Received** An annotation has been made, but it is not yet permissible to begin discussing the annotation. This is the default state of an annotation during the Fault Collection state.

2. **Open** The annotation is open for discussion. Inspection participants can add comments to the annotation's discussion thread. This is the default state of an annotation during the Discussion state of an inspection.

3. **Resolved** The problem referred to by the annotation has been resolved. The Moderator has determined that any further comments on this annotation are superfluous, and so the Moderator marks it as resolved. No further discussion of this annotation is allowed.

4. **Unresolvable** The problem referred to by the annotation cannot be resolved in the Discussion state of the inspection. Normally, this would mean that there is disagreement about the course of action to take, or that the resolution is too complicated to handle through the AnnoSpec tool. Commonly, a face-to-face meeting would be required. No further discussion of this annotation is allowed.

Note that the effects of the "Resolved" and "Unresolvable" states are identical, in that they shut off further discussion of the annotation. They differ greatly, however, in their meaning!

7.6 Summary

We have built AnnoSpec to provide a “proof-of-concept” for addressing the clutter and delocalization problems through narrow- and broad-filtering. We have described AnnoSpec at a high level. We have described the basic requirements that were informally placed upon this tool, specifically upon the version used for the pilot study reported in the next chapter. Then we have given a high-level overview of the design of the tool, and briefly touched upon some implementation issues. Finally, we described the use of AnnoSpec in detail.

7.6.1 Lessons Learned

What follows are some of the lessons we learned (or re-learned) while developing AnnoSpec.

1. Whenever one uses an application in a new way, or with a new group of users, new defects arise. Although we knew of no defects in the implementation when we released it for the experiment described in Chapter 8, our first 26 users were able to cause numerous problems.

2. Buttons should be used sparingly in a Web page. A Web page with numerous buttons is very slow to load.

3. Different browsers, even the same browser on different platforms, can give a Web page a far different look. HTML is not a good language to use for pinpoint control of a user interface.

CHAPTER 8: PILOT STUDY

We have shown that filtered annotations can be described in a mathematically consistent way. We have shown how to theoretically identify interesting foreign annotations in various types of artifacts, and how to apply filtered annotations to various types of existing systems. We have argued that an important "proof-of-concept" for the idea of filtered annotations is to create a tool that incorporates filtered annotations, and to have people use the tool and give feedback on it. We have discussed the requirements, design, and implementation of such a tool.

It remains for us to show that software professionals can apply such a tool and use foreign annotations to help them inspect an artifact. For this reason we conducted a pilot study of software inspection using AnnoSpec. Our goals were to determine the usability of AnnoSpec, and to discover whether we could usefully address clutter and delocalization during software inspection. We describe the setup of our pilot study, then report the results we achieved, and finally analyze the results with respect to our research hypotheses of Chapters 3 and 4.

8.1 Setup

8.1.1 Design of Study

26 graduate students in a software engineering course at the University of Minnesota participated in the study. All were professionals with extensive software development experience. Five reported performing software inspections as part of their work in questionnaires that they filled out. Four of these reported performing inspections manually, and one reported using a proprietary tool created by their employer, a large information technology company.

The participants were divided randomly into eight independent inspection teams (six 3-member teams and two 4-member teams). To create control groups, different teams got different versions of AnnoSpec. One version supported foreign annotations, another version supported annotating Logical Items, yet another version supported both of the above types of annotations, and the final version supported neither. Table 8.1 shows which teams had which AnnoSpec functionalities. The 4-person teams were assigned to the version of AnnoSpec with all features.

Table 8.1: Inspection teams and their functionalities.

Team ID	Nr. of Members	Supports foreign annotations?	Supports annotating logical Items?
{0, 1a, 1b}	3	No	No
{2a, 2b}	3	No	Yes
{3a, 3b}	3	Yes	No
{4a, 4b}	4	Yes	Yes

Participants reviewed code simulating a Fuel Level Monitoring system, whose requirements they had studied previously in class. They inspected 821 lines of C++ code over a 3-week period, spending the first 10 days in the Fault Collection state and the rest in Discussion. We recorded all interactions of participants with AnnoSpec.

The Producer of the artifact had left our university, so each inspection had only a Moderator and two or three Reviewers. We did not view the lack of a Producer as a significant problem. In an inspection, the Producer exists mostly to answer questions about the code being reviewed, and doesn't have the same responsibilities within an inspection as Reviewers or the Moderator. Additionally, the situation of a missing Producer is not unheard of in industry. It is unlikely that someone would leave a company in the short interval between writing code and inspecting it. However, it is reasonable that when reviewing an update to an artifact (for instance, a new version of a specification), the Producer of the original artifact may have left the company and would no longer be available to guide the Reviewers.

Participants reviewed C++ source code. We chose to have them review code because code inspections are well-understood, and C++ code has many intrinsic, well-defined Items, such as classes, attributes, methods, and statements. To understand the code, we gave the participants a copy of the system specification and design. The participants had earlier written their own specifications of this problem for their class. The specification we used was one written by the instructional staff.

Along with the specification, the participants received design information for the system. This was a high-level design, describing the various functions that would be required, but not breaking down the work into classes and methods. As part of the setup of the inspection, we gave the participants a class diagram of the system design. All the background information that we gave the participants was in hard-copy format. It was not available on-line.

8.1.1.1 Statistical Considerations

Ideally, we would like to obtain statistically significant results from any study. Thus, when setting up the pilot study, we chose a setup that would allow us to obtain such results if possible. Although it is possible to obtain significant results from experiments with software inspection (Johnson et. al., 1997), such experiments usually need to be larger than the one we are describing. Thus we did not expect to be able to obtain results with statistical significance. Nevertheless, it was easy enough to set up the pilot study so that statistically significant results could be extracted if they happened to occur. We hypothesized the existence of four factors to account for the variation in the results we would see.

The first factor was differences between the tool version with foreign annotations and that without. We hypothesized that this would show up as more annotations made among the groups with foreign annotations. Specifically, there would be more annotations made because people would view a foreign annotation and comment on it. This would add to the number of times people viewed annotations and the number of discussion comments they made. It might also affect the number of faults that people found during the Discussion phase.

The second factor was the ability to annotate Logical Items as well as lines of code. By itself, we would not expect this feature to make any difference. People who see a fault will insert it on a given line, whether or not there is a way to annotate just the faulty Item. And during Discussion, we would expect people to view annotations regardless of whether they are line annotations or logical Item annotations.

Where we feel the second factor could make a difference is in its interaction with the first factor, foreign annotations. In this case, only annotations on logical Items would be visible as foreign annotations on other lines using those Items throughout the artifacture. Without the ability to annotate logical Items, all annotations on a line containing a logical Item would have to be shown as foreign annotations on any other line in the artifacture relating to any logical Item occurring on the line. This would increase the number of “false” foreign annotations, making foreign annotations less useful. For instance, a false foreign annotation would occur if there were a line of code with a function call that contained a fault unrelated to the function. Without logical Item annotation, any annotation on this line would become a foreign annotation visible on any other line with the same function call. But in this case, the foreign annotation would be “false” because the annotation had nothing to do with the function.

Having the foreign annotation and logical Item annotation factors meant that we wanted to have four different configurations of the tool to do a full factorial experiment with these factors. Table 8.1 indicates those four configurations.

The third factor is a group effect. It is well-known that the quality of an inspection is highly dependent on the Moderator (Gilb and Graham, 1993; Humphrey, 1989). Thus the numbers of faults found, discussion comments made, and annotations viewed may well be influenced by a group dynamic of some sort. We would expect that this would not have a strong second-order effect with either foreign annotations or logical item annotations.

The final factor is the “random” factor of individual differences. Essentially, we hypothesize that any difference not accounted for by the above factors occurs because of individual differences in the inspectors, apart from the group differences. Examples of such individual differences would be familiarity with the C++ language and experience with some form of software inspection, whether or not tool-supported.

We could in theory differentiate between a group effect and an individual effect by looking at the variance of actions within a group. For instance, a Moderator who encourages each group member to thoroughly inspect the artifact will likely cause each member to make a number of annotations. In the absence of such a strong Moderator, some team members may make many annotations, and others may not.

To test generally for all the above factors would require statistical techniques suitable only for larger datasets. We can simplify our experimental design, and make it more amenable to the small amount of data we possess, by better understanding the questions that we want to ask and how they relate to the various factors. We want to ask the following questions:

1. Do some groups find more faults in the artifact than do others?
2. Do groups with logical Item annotations and/or foreign annotations have more robust discussions than groups without? We would measure this by the number of discussion comments per group and the number of annotation viewings per group.

If we assume that neither foreign annotations nor logical Item annotations impact the Fault Collection phase of inspection, then we can perform a univariate analysis of variance to answer the first question above. Answering the second question would require performing multivariate tests. Each case is complicated by the differing group sizes. We address these complications by expressing our results as per-capita numbers, at the risk of missing some effects of the group size in certain cases.

Our analysis section, Section 8.3, will discuss the statistics that we use to calculate our results, as well as discussing the results we calculate.

8.1.2 Hypotheses

From our prior experience with software inspection, and our beliefs concerning the roles of clutter and delocalization in software inspection, we developed the following hypotheses before initiating this pilot study. The first hypothesis was that during the Fault Collection phase of inspection, people would use the ability to annotate Logical Items, but that this would be of more use in classifying annotations than in helping people to discover faults.

Tool Hypothesis: *The ability to annotate logical Items will not impact the number or quality of faults collected. If annotating logical Items is prohibited, reviewers will merely annotate the line containing the Item.*

The above tool hypothesis is tangential to the major research hypotheses of this thesis. Essentially, we assume there would be no difference in fault collection rates between those with or without the ability to annotate logical Items.

The remaining hypothesis is a three-part hypothesis concerning the interaction of foreign annotations with software inspection. We motivate and state this hypothesis in Section 4.2.3 of this thesis. For convenience, we repeat the statement of the hypothesis and its two sub-hypotheses here.

Hypothesis 3: *People can make use of an annotation tool containing narrow-filtering and broad-filtering for software inspection.*

Hypothesis 3.1: *Reviewers in a software inspection will find faults during the Discussion by seeing foreign annotations.*

Hypothesis 3.2: *Reviewers in a software inspection will add to the discussion threads of foreign annotations.*

8.1.3 Data Collected

8.1.3.1 Metrics Collected

We collected eight metrics from the raw data. Our methodology for collecting the metrics was to create an ASCII record of every action of every individual, then to write Perl scripts to extract interesting portions of the record and count the number of times certain activities happened.

Six metrics were collected for each individual:

Line faults: Number of faults added by the individual to lines.

(Logical) Item faults: Number of faults added by the individual to specific logical Items. (Available to teams 2a-b and 4a-b.)

Discussion faults: Number of additional faults identified by the individual during Discussion.

Replies made: Number of replies made to annotations, other than replies in which the Moderator merely marks a fault as Resolved.

Native annotations viewed: Number of times annotations were viewed from the Item from which they were made during Discussion. During Fault Collection, people could view their own annotations, but we did not count these viewings as significant. We were looking for people to view the annotations of others and to perhaps add to a discussion thread.

Foreign annotations viewed: Number of times foreign annotations were viewed. (Available to teams 3a-b and 4a-b.)

Two metrics were collected on a per-team basis:

Faults resolved: Number of faults resolved.

Faults unresolvable: Number of faults marked as unresolvable.

Some faults were neither resolved nor left unresolvable, but left open because the server crashed for 12 hours shortly before the inspection was due to end. Afterward, the Moderators used their judgement whether to mark faults as unresolvable or to leave them open.

Additionally, we analyzed the detailed logs of the inspections to identify instances of viewing a foreign annotation. We sought replies added from foreign annotations and faults added during discussion that appeared motivated by a foreign annotation previously viewed.

8.1.3.2 Questionnaires and Other Feedback

We gave people two questionnaires to answer during the pilot study. We gave them one questionnaire after Fault Collection and another after Discussion. These questionnaires form Appendices 1 and 2 of this thesis, respectively.

The questionnaires were used to probe the qualitative responses of the people to the tool, allowing us to learn things that the metric data would not tell us, such as the overall feeling of whether the tool was useful or not.

The questionnaires were nominally anonymous, although people were required to give the number from their group ID to let us know which version of the tool they were using. Thus, any given questionnaire could be known to have been answered by one of six people. Additionally, some people chose to sign their names, eliminating any anonymity.

8.2 Results

8.2.1 Data Collected

About four hours before the inspection was due to end, the server hosting the inspection crashed for 12 hours. This prevented some people from adding annotations that might have resolved faults, and prevented Moderators from marking more faults as resolved.

Moreover, there were four updates made to the tool during the first five days of its 3-week use. These did not affect the usability of the tool, but they were put in to fix errors that caused occasional tool crashes.

Table 8.2 shows the raw data collected, grouped by inspection team. Table 8.3 shows the raw results for each individual, noting their inspection team. It includes the rank of each individual in number of fault collected, number of discussion comments, and number of annotations viewed. Table 8.4 shows the per-person statistics, combined by type of AnnoSpec tool.

Table 8.2: Per-team summary data. "N/A" means the feature was unavailable to this team.

ID	1a	1b	2a	2b	3a	3b	4a	4b	SUM
Line Faults	22	36	54	20	92	31	44	60	**359**
Item Faults	N/A	N/A	13	0	N/A	N/A	2	11	**26**
Tot. Faults	22	36	67	20	92	31	46	71	**385**
Discussion faults	0	0	2	0	3	2	0	10	**17**
Native Annos Viewed	62	89	655	69	471	89	210	992	**2637**
Foreign Annos Viewed	N/A	N/A	N/A	N/A	13	20	30	35	**98**
Replies	3	2	48	0	21	19	19	21	**133**
Faults Left Open	18	30	18	3	76	31	29	37	**242**
Faults Resolved	4	6	43	16	14	0	17	30	**130**
Faults Unresolvable	0	0	6	1	2	0	0	4	**13**

Table 8.3: Per-person summary data.

Group ID	Line Annos	File Annos	Log. Annos	*Total Annos*	*Anno Rank*	Disc. Comm	*Disc Rank*	View Native	View Fgn.	*Total Views*	*View Rank*	*Total Acts.*	*Sum Ranks*	*Rank Sum*
1a	6	0		*6*	*22*	1	*16*	5		*5*	*23*	*12*	*61*	*3*
1a	9	2		*11*	*14*	1	*16*	21		*21*	*20*	*33*	*50*	*8*
1a	5	2		*7*	*19*	1	*16*	36		*36*	*17*	*44*	*52*	*6*
1b	24	0		*24*	*5*	0	*19*	3		*3*	*24*	*27*	*48*	*9*
1b	7	0		*7*	*19*	2	*14*	46		*46*	*14*	*55*	*47*	*11*
1b	5	0		*5*	*24*	0	*19*	40		*40*	*16*	*45*	*59*	*4*
2a	26	4	10	*40*	*2*	15	*4*	175		*175*	*5*	*230*	*11*	*25*
2a	15	2	2	*19*	*8*	16	*3*	322		*322*	*2*	*357*	*13*	*24*
2a	6	1	1	*8*	*18*	17	*1*	158		*158*	*6*	*183*	*25*	*20*
2b	1	0	0	*1*	*25*	0	*19*	2		*2*	*25*	*3*	*69*	*2*
2b	12	0	0	*12*	*12*	0	*19*	17		*17*	*22*	*29*	*53*	*5*
2b	7	0	0	*7*	*19*	0	*19*	50		*50*	*13*	*57*	*51*	*7*
3a	41	0		*41*	*1*	4	*12*	178	4	*182*	*4*	*227*	*17*	*23*
3a	21	10		*31*	*3*	0	*19*	19	2	*21*	*20*	*52*	*42*	*13*
3a	16	4		*20*	*6*	17	*1*	274	7	*281*	*3*	*318*	*10*	*26*
3b	0	0		*0*	*26*	0	*19*	0	0	*0*	*26*	*0*	*71*	*1*
3b	9	2		*11*	*14*	11	*5*	47	7	*54*	*12*	*76*	*31*	*16*
3b	17	3		*20*	*6*	8	*7*	42	13	*55*	*11*	*83*	*24*	*21*
4a	12	2	0	*14*	*11*	3	*13*	81	9	*90*	*8*	*107*	*32*	*15*
4a	10	0	1	*11*	*14*	2	*14*	22	1	*23*	*18*	*36*	*46*	*12*
4a	8	0	1	*9*	*17*	8	*7*	36	6	*42*	*15*	*59*	*39*	*14*
4a	12	0	0	*12*	*12*	6	*9*	71	14	*85*	*9*	*103*	*30*	*17*
4b	17	0	2	*19*	*8*	5	*11*	57	5	*62*	*10*	*86*	*29*	*18*
4b	6	0	9	*15*	*10*	0	*19*	20	2	*22*	*19*	*37*	*48*	*9*
4b	3	3	0	*6*	*22*	11	*5*	767	22	*789*	*1*	*806*	*28*	*19*
4b	31	0	0	*31*	*3*	6	*9*	148	6	*154*	*7*	*191*	*19*	*22*

Table 8.4: Per-person summary statistics. (mean ± standard deviation)

Team	**1a,b**	**2a,b**	**3a,b**	**4a,b**
Faults Collected	10.0 ± 6.5	14.5 ± 12.6	20.5 ± 13.2	14.6 ± 7.2
Replies	0.8 ± 0.7	8.0 ± 8.0	6.7 ± 6.1	5.1 ± 3.3
Native Annos Viewed	25.2 ± 16.8	120.7 ± 111.6	93.3 ± 99.0	150.3 ± 236.3
Foreign Annos Viewed	N/A	N/A	5.5 ± 4.2	8.1 ± 6.5

8.2.2 Questionnaire Responses

Of the 26 participants in the inspections, we got 25 questionnaires back after both the Fault Collection and Discussion phases of inspection. Table 8.5 reports the amount of time people stated that they spent on the inspection. We could not break this down by team because some people did not properly report their team ID. One point that came out of the questionnaires was that people who had modern development tools at their disposal (such as Microsoft Visual Studio) would download the code to be inspected and browse it using that tool, not within the web browser. We discuss the other results of the questionnaires in our Analysis section.

Appendix 1 gives selected responses of participants to the Fault Collection questionnaire. Appendix 2 gives selected responses of participants to the Discussion questionnaire. Responses were selected for inclusion based on their content. Thus, we did not include responses such as "good tool" or "the code is awful". Since some responses were very long, they have been elided where noted with ellipses.

Table 8.5: Time spent on inspection. In hours. Not grouped by team because some responses did not properly indicate team ID.

Phase	Mean Time	Std.Dev. of Time
Fault Collection	3.9	1.2
Discussion	2.3	1.2

8.3 Analysis

At a high level, we successfully incorporated narrow-filtering to reduce clutter and broad-filtering to address delocalization in a tool for distributed, asynchronous software inspection, meeting our most basic goal. People used the tool to inspect an industrial-size software artifact.

We divide our detailed analysis into three parts. We analyze the metrics that we collected, then we analyze the results of the questionnaires that we distributed and of the e-mails that we received, and finally we discuss our hypotheses in light of our previous analyses.

8.3.1 Data Analysis

The wide variability of the data among inspections precludes our arriving at statistically significant conclusions concerning our hypotheses. We will analyze the data for number of annotations made, number of discussion comments made, and number of annotations viewed to look at the variability.

8.3.1.1 Number of Annotations Made

The number of faults collected per person (Figure 8.1) had a mean of 14.9 and a standard deviation of 11.0. A normal quantile-quantile plot (Figure 8.2) suggests that the data are not normal. Since we have a small sample size of only 26 observations, the lack of normality makes any hypothesis testing for equality of means suspect, because we cannot invoke the central limit theorem with such a small sample size.

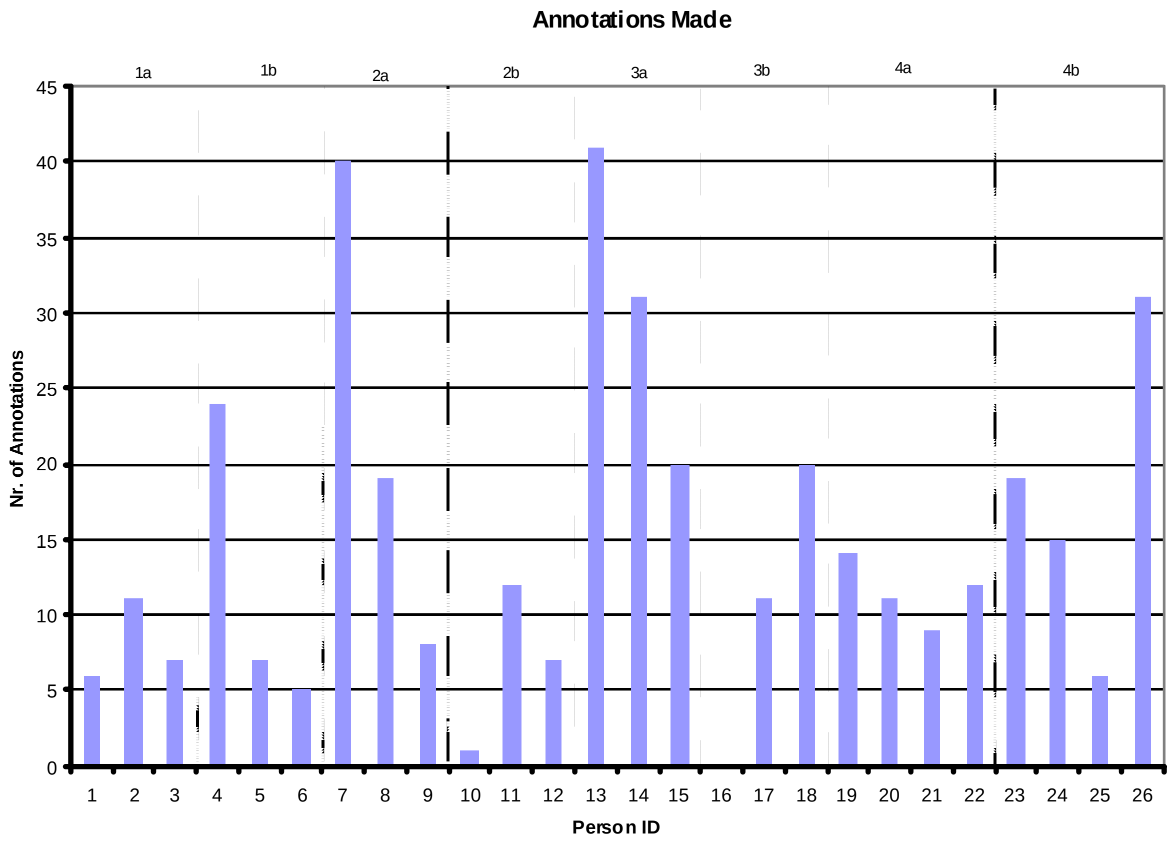


Figure 8.1: Number of annotations per person. Group IDs are shown.

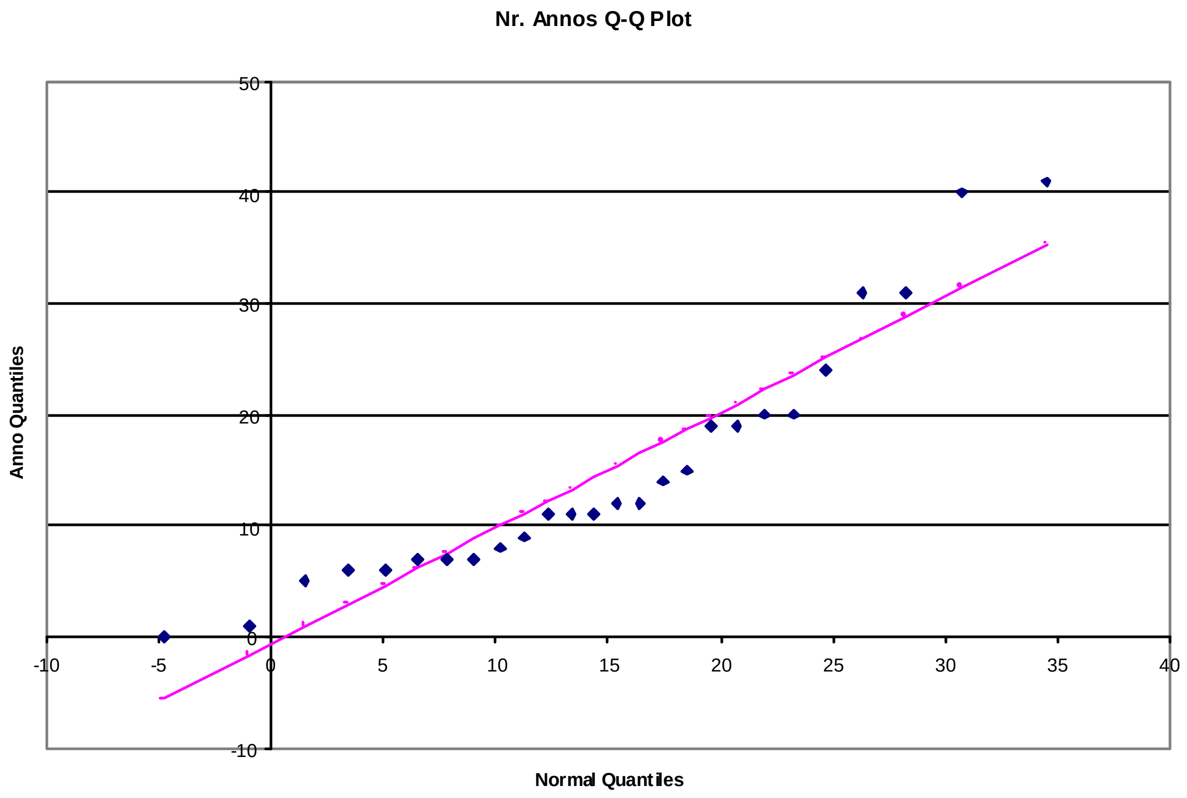


Figure 8.2: Quantile-Quantile plot of annotations vs. anticipated values for a normal distribution. The "bowing" of the annotation plot suggests that the data are not normally distributed. Specifically, the "tail" of the distribution is too short on the low end, and too long on the high end.

Table 8.6: Statistics for per-capita annotations, reported by inspection group. "StdDev" is the standard deviation, and "CoV" is the coefficient of variation.

Group	1a	1b	2a	2b	3a	3b	4a	4b
Size	3	3	3	3	3	3	4	4
Mean	8.0	12.0	22.3	6.7	30.7	10.3	11.5	17.8
StdDev	2.6	10.4	16.3	5.5	10.5	10.0	2.1	10.4
CoV	0.3	0.9	0.7	0.8	0.3	1.0	0.2	0.6

We wanted to determine whether there was a statistically significant difference in the mean number of annotations added as a result of allowing people to annotate logical Items. The mean number of annotations in groups without logical Items was 12.3, with a standard deviation of 10.8, and a sample size of 12. The mean number of annotations in groups with logical Items was 17.1, with a standard deviation of 11.0, in a sample size of 14. Although the mean number of annotations per person in the groups with logical items is about 40% higher than the mean for people with no ability to annotate logical Items, the standard deviations are so large that the results are not statistically significant.

Another statistical question is whether there is a group effect. Table 8.6 shows the means and standard deviations of the numbers of annotations made per person for each group.

Because of the small sample sizes and the wide variation we cannot make any statistical claims to differences in the mean numbers of annotations made. But we can suggest that there may be a group effect. This group effect could come about because of a strong Moderator urging people to collect faults. We can see no way to select between these two alternatives given our data.

One additional point arises from our analysis of the text of the annotations made. People who could annotate either a line or a logical Item were inconsistent in choosing when to annotate a line, and when to annotate a logical Item. Some people would attach a fault to a line, and others would attach the same fault to an Item on that line. Thus it appears that people must be trained to select the proper scope for an annotation, if they are to be given a choice.

8.3.1.2 Discussion Comments Made

Figure 8.3 is a plot of the number of discussion comments made per person. There was a mean of 5.1 comments per person, with a standard deviation of 5.8. The high standard deviation occurred largely because 8 people made no substantive discussion comments, even though all but one person viewed annotations. We cannot see any way to extract any statistically significant information about foreign annotations or logical Item annotations from such data.

Table 8.7 gives some statistics concerning Discussion comments. Little can be gleaned from these statistics, largely because the number of Discussion comments was so low.

Notice especially the two groups 2a and 2b, who used the identical version of the tool. One group had the most discussion comments of any group, and the other group had no discussion comments at all!

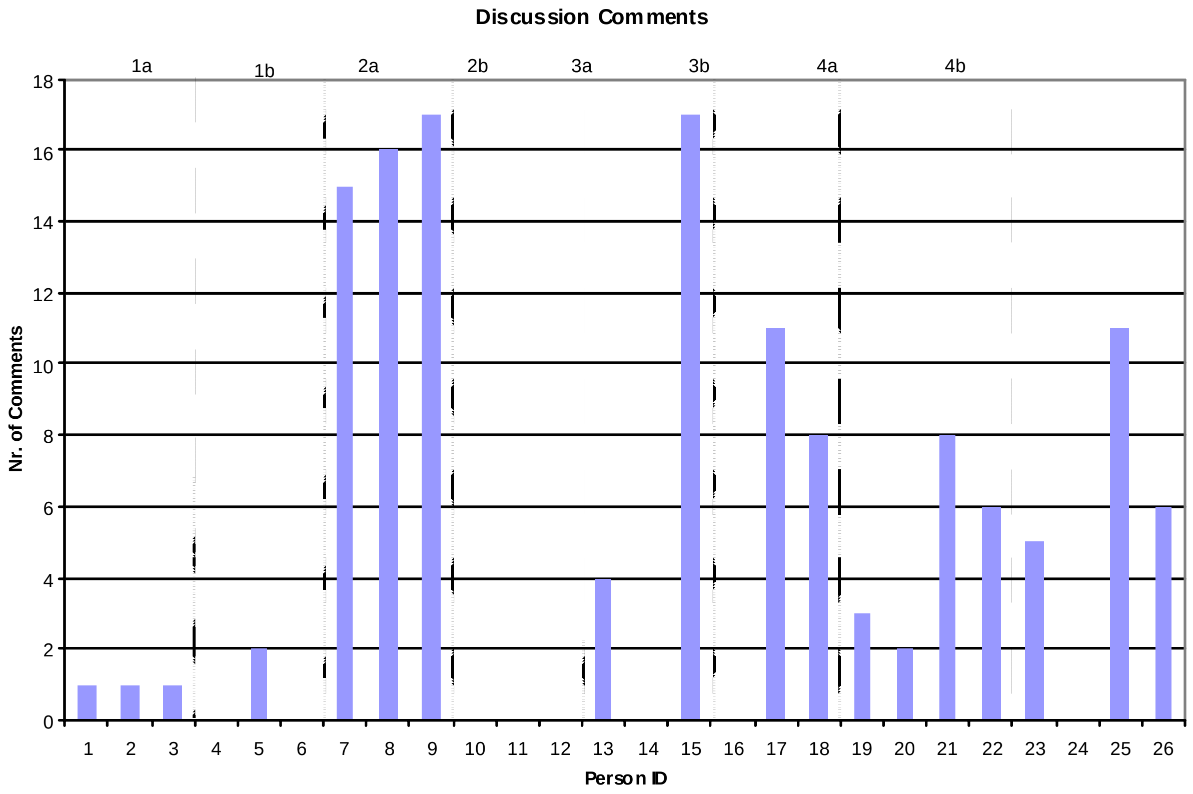


Figure 8.3: Number of discussion comments made per person, with group IDs shown.

Table 8.7: Statistics for per-capita discussion comments, reported by inspection group. "StdDev" is the standard deviation, and CoV is the coefficient of variation.

Group	**1a**	**1b**	**2a**	**2b**	**3a**	**3b**	**4a**	**4b**
Size	3	3	3	3	3	3	4	4
Mean	1.0	0.7	16.0	0.0	7.0	6.3	4.8	5.5
StdDev	0.0	1.2	1.0	0.0	8.9	5.7	2.8	4.5
CoV	0.0	1.7	0.1	0.0	1.3	0.9	0.6	0.8

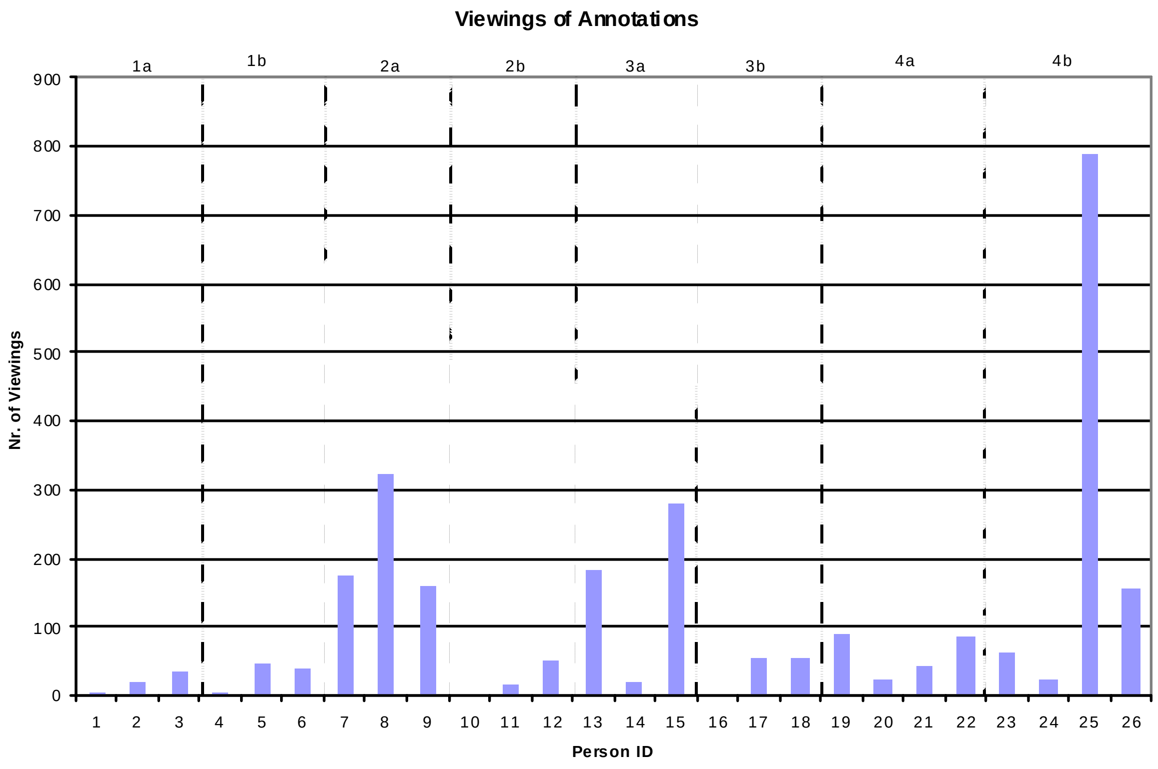


Figure 8.4: Number of annotations viewed per person, including both native and foreign annotations.

8.3.1.3 Numbers of Annotations Viewed

The number of annotations viewed per person also varied widely. In Figure 8.4 we have a single, large outlier, which would skew the mean results for group 4b and for the dataset as a whole. Nevertheless, for completeness we report the results in Table 8.7. The global mean number of annotations viewed was 105.2, with a standard deviation of 163.3.

Table 8.8: Statistics for per-capita annotations viewed, reported by inspection group. "StdDev" is the standard deviation, and CoV is the coefficient of variation.

Group	1a	1b	2a	2b	3a	3b	4a	4b
Size	3	3	3	3	3	3	4	4
Mean	20.7	29.7	218.3	23.0	161.3	36.3	60.0	256.8
StdDev	15.5	23.3	90.2	24.6	131.2	31.5	32.8	359.1
CoV	0.8	0.8	0.4	1.1	0.8	0.9	0.5	1.4

One observation, which serves as sort of a "reality check", is to look at the correlations among the per capita number of annotations made, number of discussion comments made, and number of annotations viewed. This is shown in Table 8.9. Table 8.9 suggests the obvious result, that the more annotations one views, the more discussion comments one makes. This is not an interesting result in itself, but it serves to suggest that at least in some ways, the data seem to make sense.

Table 8.9: Correlation coefficients among data measured.

First Item	Second Item	Correlation Coefficient
Annotations Made	Discussion Comments	0.29
Annotations Made	Annotations Viewed	0.40
Discussion Comments	Annotations Viewed	0.82

Another question we can ask is the degree to which participants in groups 3{a,b} and 4{a,b} used foreign annotations. Overall, these participants viewed foreign annotations 53 times, thus averaging one view of a foreign annotation per every 18.0 views of a native annotation. That is, 5.3% of annotations viewed were viewed as foreign annotations.

If foreign annotations were useful, people would add to the discussion threads of annotations that they viewed as foreign annotations. Were people to only add to discussion threads of annotations that they viewed as native annotations, we would surmise that people did not find foreign annotations as useful as native annotations, because they didn't use foreign annotations to add information to the inspection.

From our data above, if viewing foreign annotations were as useful to participants as viewing native annotations, we would expect about 5.3% of discussion comments in groups with foreign annotation capabilities would be made to foreign annotations. That is, people would make a discussion comment on an annotation that they viewed as a foreign annotation without taking any intervening action. Given that people in these groups made 80 discussion comments, we would have expected them to make about 4 comments off foreign annotations (4.2 to be exact).

We observed that people in groups 3 and 4 made five discussion comments off foreign annotations. Thus some participants found foreign annotations useful.

Teams with foreign annotations found 15 faults during Discussion. We would expect participants who discovered a fault because of viewing a foreign annotation to add a new fault, possibly on a different file, immediately after viewing that foreign annotation. In two instances, a participant viewed a foreign annotation, then immediately placed a new annotation on an Item. In one case, the annotation was placed on an Item related to the originally annotated Item in another file; in the other, the annotation was placed on the Item from which the foreign annotation was viewed.

Statistically, we would have expected 5.3% of the 15 Discussion faults (0.8 fault) to be added after viewing foreign annotations. Finding two such annotations qualitatively supports our hypothesis.

This inspection differed from previous distributed, asynchronous inspections that our research group has done (Mashayekhi, 1995; Stein et. al., 1997) in that there were relatively few replies to annotations. This may be partially due to the nature of the artifact. This study involved a code inspection, and many of the errors found were straightforward (e.g., incorrect use of case statement). Our previous reviews have been of designs, whose errors can be less straightforward and so engender more discussion.

The inspected artifact was poorly written and too long. 821 lines of C++ code was too much, even though the code was simple. The code failed to implement many requirements, and it was poorly commented. In hindsight, we should have cleaned up the artifact prior to inspection, and followed recommended guidelines for amount of code to place into one inspection (Stein et. al., 1997; Christensen et. al., 1993).

8.3.2 Questionnaire Analysis

Appendices 1 and 2 give the results of the questionnaires. From these questionnaire results, we can make the following observations.

1. Our pilot study was compromised by the tool's being very slow at times. Some of the pilot study participants were experienced Web performance analysts. They all said that the tool would have been much faster had we replaced many of the buttons by hyperlinks. As it was, a number of people didn't use the tool at all to browse the code, but loaded the code into an unrelated development system to browse.

   This browsing was confirmed by additional e-mail messages that people sent, indicating that they would like to browse the code off-line and asking if there was any reason not to do so.

2. Many people complained that the buttons for making and viewing annotations took up too much screen real estate, both vertically (too few lines of code were shown) and horizontally (not all of each line could be seen). This problem was especially severe with the Netscape browser used under the Windows operating system. A solution recommended to this problem in some e-mails was be to use hyperlinks instead of buttons for making and viewing annotations. Notice that this solution also solves the first problem pointed out above.

3. People generally liked the idea of being able to annotate logical Items in addition to lines of code. 13 of the 15 people who responded to questions 6 and 7 on the Fault Resolution questionnaire thought the idea was a good one. Only people who had a version of a tool that allowed annotation of logical Items were asked to respond to this question. Oddly, 14 people had the ability to annotate logical Items, but 15 people responded to the question.

4. People were decidedly mixed in their liking for foreign annotations. Of the 12 people who responded to the question, 7 thought the idea of foreign annotations had merit, while 5 thought the idea of using foreign annotations to be of little or no value.

In summary, the questionnaires brought out some serious flaws with our pilot study:

- The inspection was too large.
- The inspected material was too defect-laden.
- The user interface of the tool had some problems, including performance.
- The server hosting the tool was unreliable.
- The tool itself had some defects that impacted fault collection.

8.3.3 Analysis with Respect to Hypotheses

In this section, we discuss our results in light of our research hypotheses.

Tool Hypothesis: *The ability to annotate logical Items will not impact the number or quality of faults collected. If annotating logical Items is prohibited, reviewers will merely annotate the line containing the Item.*

This is essentially a “null” hypothesis, that the inclusion of a tool feature makes no difference by itself. To argue against this hypothesis, we would need to see that some types of faults were found with much higher frequency by people with logical Item annotations, and that these people associated these faults with the logical Items.

The number of faults discovered per person varied much more within an inspection group than between inspection groups, so we cannot make any quantitative claims about this hypothesis.

However, people were inconsistent in assigning the same fault to a line or a logical Item. In teams with the ability to annotate logical Items, sometimes one person would associate a fault with a logical Item while another member of the same team would associate the same problem with a line. Thus, it seems unlikely that being able to annotate logical Items would enhance the number or quality of faults seen, because even when it was available people used it inconsistently.

Yet people who had the ability to annotate logical Items almost unanimously liked having the ability to do so. Thus it appears that the idea has merit, but must be implemented carefully.

Hypothesis 3.1: *Reviewers in a software inspection will find faults during the Discussion by seeing foreign annotations.*

As described above, reviewers added two faults during the Discussion phase immediately after viewing a foreign annotation on an Item, where the faults were related to the foreign annotation. This is a reasonable number for the size of the inspection, and suggests that Reviewers can find faults during Discussion that are suggested to them by reading foreign annotations.

Hypothesis 3.2: *Reviewers in a software inspection will add to the discussion threads of foreign annotations.*

Participants in teams with foreign annotations averaged one view of a foreign annotation per every 18.0 views of a native annotation, for 5.3% of total annotation viewings. These participants made 80 replies to annotations, five of which came off of foreign annotations. These frequencies of activities after viewing foreign annotations are consistent with the frequencies of comparable activities after viewing native annotations. Thus Reviewers will add to the discussion threads from foreign annotations.

8.4 Summary

We performed a proof-of-concept pilot study with the AnnoSpec tool to demonstrate that people could perform software inspection with a tool that supported either or both of annotations on logical Items and foreign annotations. People with all combinations of these features were able to perform software inspection of a C++ source code artifact by finding faults and resolving them in an asynchronous, distributed manner.

People were enthusiastic about the ability to annotate logical Items, but in practice they had a hard time applying the concept.

People had mixed feelings about the utility of showing foreign annotations, but some people found them worthwhile, and they effectively used foreign annotations during the Discussion phase of inspection.

Few concrete results can be obtained from a study of this size. But some additional flaws in this study, such as the server problem and the too-great volume of inspection material, also contributed to a lack of concrete results.

CHAPTER 9: CONCLUSION

Annotation of software artifacts is an important software development activity. Various artifacts are subject to annotation, including requirements specifications, designs, source code, test plans, and project plans – in general, any human-readable element of the artifacture of software development. Artifacts may be annotated in at least two situations. They may be annotated while being written, especially if the documents are collaboratively written. Or they may be annotated during informal review or formal software inspection.

In this thesis, we have identified that the delocalization problem pertains to annotations, and we have described the interaction of the problem of delocalized annotations with the better-known problem of clutter. We have developed a graph-theoretic model of annotation and illustrated some applications of this model. Using this model we have built an annotation and inspection tool, AnnoSpec, that incorporates filtering of annotations to concurrently address the problems of clutter and delocalization. Using AnnoSpec, groups of software professionals were able to perform distributed, asynchronous software inspections.

9.1 Contributions

This thesis makes three contributions to the state of the art of annotating software systems.

4. It taxonomizes three dimensions along which delocalization occurs within annotated artifacts: lateral, longitudinal, and historical delocalization.
5. It develops a mathematical model of annotated artifacts that allows us to express annotation visibility along various axes.
6. It verifies that a system can be built to ameliorate delocalization, and can be used by software professionals in formal software inspection.

9.1.1 Taxonomy of Delocalization

It is useful to extend annotation visibility in at least three ways to address delocalization issues. Our first contribution is to describe these axes of delocalization.

We may extend annotation visibility to other portions of a specific annotated artifact, thus addressing *lateral* delocalization. One example of lateral delocalization is when a programmer consistently calls the wrong function to perform a task in a program. An inspector may discover this problem at any occurrence of this error, yet the annotation is valid to all occurrences of the error. Thus the programmer should be able to see the annotation while viewing any occurrence of that error.

We may extend annotation visibility to other portions of the artifacture of a software system, thus addressing *longitudinal* delocalization. For instance, if a customer annotates a requirement in a specification to explain the meaning of that requirement, that annotation may be of interest to developers who are designing or implementing a system to meet that requirement.

We may extend annotation visibility to earlier or later versions of an annotated artifact, thus addressing *historical* delocalization. The inspected version of an artifact is not the one that is ultimately produced. Rather, the inspected version is modified to address concerns raised during the inspection. These annotations may be of interest to those who are looking at later versions of the artifact, because they explain the rationale behind certain decisions.

In Chapter 3, we described the above types of annotation visibility in detail. We recommended ameliorating delocalization by using *foreign annotations*: annotations on one part of the artifacture that are made visible to viewers of some other part of the artifacture. Having foreign annotations increases the number of annotations that a viewer may see when looking at a software artifact. This can lead to *clutter,* in which the user sees too many annotations. In Chapter 4, we considered in detail the motivating example for our work, formal software inspection. We described a protocol for distributed, asynchronous software inspection that is amenable to addressing issues of clutter and delocalization of annotations.

9.1.2 Graph-theoretic Model of Annotation

It is important to qualitatively describe the clutter and delocalization problems, as well as enhancements to annotation visibility, so that people can understand the problems and their resolution. However, such a qualitative description is not sufficient to allow us to build a system to address delocalization. Thus a second contribution of our work was the construction of a graph-theoretic model of annotation. Chapter 5 defined this model, and proved some theorems that, taken together, indicate that we can use our model to filter annotations to users, and by this filtering address the clutter and delocalization issues simultaneously. Appendix 3 is an alternative formulation of this model in the Z specification language. Chapter 6 gave some theoretical examples of applying this model to hypothetical artifacts and existing systems.

9.1.3 System for Addressing Delocalized Annotations

Chapters 7 and 8 showed that our approach of filtering to address clutter and delocalization is workable in practice. We described the requirements, design, and use of the AnnoSpec tool to address clutter and delocalization in software artifacts. We also described the results of a pilot study of inspection we performed on C++ source code using AnnoSpec. This pilot study suggested that filtering may be a useful way to deal with clutter and delocalization in annotated artifacts during formal software inspection.

9.2 Summary of Research Hypotheses

Once we had developed the idea of filtering to address clutter and delocalization, we came up with three research hypotheses concerning such filtering. We reproduce these hypotheses and their sub-hypotheses here, and discuss them in light of our results.

Hypothesis 1: *Broad-filtering can be used to successfully ameliorate delocalization.*

> **Hypothesis 1.1:** *Broad-filtering can be described mathematically in such a way that it can be applied clearly and unambiguously.*

> **Hypothesis 1.2:** *Broad-filtering can be implemented in an annotation tool.*

We demonstrated the successful use of broad-filtering to ameliorate delocalization, in theory and in practice. In Chapter 3, we described broad-filtering to address delocalization and argued for its usefulness. In Chapter 5, we demonstrated how to represent broad-filtering as a set of neighborhoods of nodes in a graph of an annotated software system. In Chapter 6 we illustrated clear, unambiguous applications of broad-filtering. In Chapter 7 we described how to implement broad-filtering in an annotation tool. During the pilot study reported in Chapter 8, professionals were able to use the broad-filtering aspects of the tool to see desired foreign annotations.

Hypothesis 2: *Narrow-filtering can be successfully combined with broad-filtering to simultaneously reduce clutter and ameliorate delocalization.*

> **Hypothesis 2.1:** *Simultaneous narrow- and broad-filtering can be described mathematically in such a way that they can be applied concurrently without ambiguity.*

> **Hypothesis 2.2:** *Simultaneous narrow- and broad-filtering can be implemented in an annotation tool.*

Our work in this thesis demonstrated the successful integration of narrow-filtering with broad-filtering to manage clutter while ameliorating delocalization. In Chapter 5 we described how to superimpose narrow-filtering and broad-filtering. In Chapter 6, we provided theoretical examples of performing this concurrent narrow-filtering and broad-filtering. In Chapter 7 we described how to implement narrow- and broad-filtering simultaneously. In our study in Chapter 8, professionals were able to implement narrow- and broad-filtering concurrently.

Hypothesis 3: *People can make use of an annotation tool containing narrow-filtering and broad-filtering for software inspection.*

> ***Hypothesis 3.1:*** *Reviewers in a software inspection will find faults during the Discussion by seeing foreign annotations.*
>
> ***Hypothesis 3.2:*** *Reviewers in a software inspection will add to the discussion threads of foreign annotations.*

Hypothesis 3 concerns itself with the practical usefulness of filtering. Our analysis of Hypothesis 3 is based on the results of our pilot study reported in Chapter 8. Because of the small size of the study, we were unable to draw firm statistical conclusions. Moreover, the differences between the results of inspections performed by different teams of people on the same artifact were overwhelming, even among teams using the same version of the AnnoSpec tool. So we can make no firm statement about the validity of Hypothesis 3. It was not clear from our limited study whether the benefits of broad-filtering are worth the added complexity.

Some people used filtered annotations effectively during software inspection; in their questionnaires they wrote that they thought the concept of broad-filtering had merit. But almost as many people were confused by the concept of broad-filtering as liked it. Either they thought the concept itself was of no utility, or they felt that the complications it introduced into an inspection tool overwhelmed any positive benefits to be derived by broad-filtering annotations.

9.3 Future work

We see four possible directions for future work on filtered annotations. We could extend the taxonomy of annotations, extend the annotation model, enhance the AnnoSpec system, or find other applications of filtered annotations.

9.3.1 Taxonomy

In this thesis, we have taxonomized delocalization of annotations along three axes – lateral, longitudinal, and historical. These axes seem to be natural axes for delocalization within software artifacts, but there might be others. Either an extension to our taxonomy, a proof of the completeness of our taxonomy as it stands, or an extension followed by a proof of completeness of the extended taxonomy would be a significant research contribution.

9.3.2 Graph-theoretic Model

We have also described a mathematical model of annotated artifacts. One area of of future work would be to identify and prove additional theorems about this model, thus extending the model. Another possible extension would be to apply it to more systems than those shown in this thesis, beyond inspection systems and collaborative authoring systems.

9.3.3 AnnoSpec System

The practicality of using filtered annotations has not yet been determined, partially because there were significant flaws in the user interface and performance of the AnnoSpec tool. One direction for future work would be to improve the interface of AnnoSpec, so that it becomes practically useful, and so that people will actually review artifacture online, within the tool.

One possible form of this work would be to integrate AnnoSpec with an existing development environment. This would integrate the addressing of historical delocalization problems into the development environment. Such integration is probably necessary for effectively seeing annotations from previous versions of a software artifact.

Given a better tool and/or tool interface for supporting filtered annotations, it would become possible to run further experiments which might obtain statistically significant results concerning the usefulness of filtered annotations during software inspection.

9.3.4 New Applications of Filtered Annotation

Other work could explore more applications of filtered annotations beyond software inspection.

People could use filtered annotations while collaboratively writing a document. Annotation systems are already embedded in some word processors. One direction for future work would be to add filtered annotations to a simple, public-domain word processing tool, and see whether filtered annotations make collaborative writing easier.

Instructors could use filtered annotations in teaching as a way to grade assignments, especially programming assignments. Grading a program is basically a form of software inspection. The actual grading is tantamount to Fault Collection. This is often followed by a "Discussion", in which the student visits the instructional staff to understand the annotations made, or to argue about whether a certain annotation respresents a real fault. In our experience, students often repeatedly make the same mistakes thorughout a program; this could make foreign annotations a convenient way to grade assignments. Storing the assignment under a database with sufficient privacy guarantees could also reduce the incidence of some types of cheating, and would give the instructional staff ready access to the student's work when assigning course grades.

BIBLIOGRAPHY

Baldwin, J. 1992. An Abbreviated C++ Code Inspection Checklist. University of Illinois Technical Report, available from http://www.ics.hawaii.edu/~johnson/FTR/Bib/Baldwin92.html.

Ballman, K., and Votta, L. 1994. Organizational Congestion in Large-Scale Software Development. *3rd International Conference on the Software Process,* October 1994.

Berman, E. 1998. Software Inspections at Fermilab - Use and Experience. *IEEE Transactions on Nuclear Science*, Vol. 45, Nr. 4, August 1998.

Bessler, S., Hager, M., Benz, H., Mecklenburg, R., and Fischer, S. 1997. DIANE: A Multimedia Annotation System. *Proceedings of the 2nd European Conference on Multimedia Applications, Services, and Techniques (ECMAST'97).*

Bingler, R., Albrecht, S., Bachrach, M., Carnell, S., Ratliff, M., Pitti, D., Unsworth, J., and Vande Creek, D. Inote: An Image Annotation Tool in Java. *Institute for Advanced Technology in the Humanities Technical Report*, http://jefferson.village.virginia.edu/inote.

Bly, S., Harrison, S., and Irwin, S. 1993. Media Spaces: Bringing People Together in a Video, Audio, and Computing Environment. Communications of the ACM, Vol. 36, Nr. 1, January 1993.

Brade, K., Guzdial, M., Steckel, M., and Soloway, E. 1994. Whorf: A Hypertext Tool for Software Maintenance. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 4, Nr. 1.

Brothers, L., Sembugamoorthy, V., and Miller, M. 1990. ICICLE: Groupware for Code Inspection. *Proceedings of Computer Supported Cooperative Work,* October 1990.

Cavalier, T. Chandhok, R., Morris, J., Kaufer, D., and Neuwirth, C. 1990. A Visual Design for Collaborative Work: Columns for Commenting and Annotation. *Proceedings of HICCS'24*.

Chernak, Y. 1996. A Statistical Approach to the Inspection Checklist Formal Synthesis and Improvement. *IEEE Transactions on Software Engineering*, Vol. 22, Nr. 12, December 1996.

Christenson, D., Huang, S., Lamperez, A., Smith, D. 1993. Statistical Methods Applied to Software. *Total Quality Management for Software,* pp. 351-386 , G. Schulmeyer and J. McManus, eds., Van Nostrand Reinhold.

Columbia University LibraryWeb. http://www.columbia.edu/cu/libraries/indiv/rare/images/date.html. This specific image is http://www.columbia.edu/cu/libraries/indiv/rare/images/img0019/medium.jpg

Corriveau, J., and Hayashi, C. 1994. A Strategy for Realizing Traceability in an Object-Oriented Design Environment. *Proceedings 4th International Workshop on Computer Aided Systems Theory (CAST'94),* May 1994.

Cubus Corp., http://www.cubus.net.

Date, C. 1977. *An Introduction to Database Systems,* 2nd Ed., Addison-Wesley 1977.

Dewan, P., and Choudhury, R. 1994. Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Information Systems,* December 1994.

Diaper, D., and Beer, M. 1995. Collaborative Document Annotation using Electronic Mail. *Computer Supported Collaborative Work,* Vol. 3, Nr. 3-4, 1995.

Diller, A. 1994. *Z: An Introduction to Formal Methods,* John Wiley & Sons.

Dynatext. http://www.inso.com/dynatext

Fagan, M. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal,* Vol. 15, Nr. 3.

Fowler, M. 1997. UML Distilled, Addison-Wesley.

Furata, R., and Stotts, P. 1994. Interpreted Collaboration Protocols and Their Use in Groupware Prototyping. *Proceedings of CSCW'94*, 1994.

Library of Congress. "Gettysburg Address" web site, lcweb.loc.gov/exhibits/gadd/gadrft.html

Indiana University. Department of Computer Science web site. www.cs.indiana.edu/statecraft/gettysburg.html

Gilb, Tom, and Graham, D. 1993. *Software Inspection,* S. Finzi (ed.) Addison-Wesley, 1993.

Gilb, Tom, Gilb, Kai, and Gilb, Tor. 1998. *Inspection Team Leadership Workshop,* available from www.Result-Planning.com.

Gilb, Tom. 1999. Software Inspections are Not for Quality, but for Engineering Economics. *draft of article to appear in IEEE Software,* available from www.Result-Planning.com.

Gintell, J., Arnold, J., Houle, M., Kruszelnicki, J., McKenney, R., and Memmi, G. 1991. Scrutiny: A Collaborative Inspection and Review System. Proceedings of the 4th European Software Engineering Conference, September 1991.

Gintell, J., and Memmi, G. 1992. CIA: Collaborative Inspection Agent Experience: Building a CSCW Application for Software Engineering. Workshop on CSCW Tools, October 1992.

Goeschka, K., Falb, J., and Radinger, W. 1998. Database Access with HTML and Java – A Comparison Based on Practical Experience. *Proceedings of the 22nd Annual International Computer Software and Applications Conference (COMPSAC'98),* August 1998.

Gotel, O., and Finkelstein, A. 1994. An analysis of the requirements traceability problem. *Proceedings of the First International Conference on Requirements Engineering*. 1994.

Gotel, O., and Finkelstein, A. 1995. Contribution Structures. *2nd IEEE International Symposium on Requirements Engineering,* March 1995.

Gotel, O., and Finkelstein, A. 1996. Revisiting requirements production. *Software Engineering Journal,* vol.11, no.3, May 1996.

Humphrey, W. 1989. *Managing the Software Process,* Addison-Wesley.

Johnson, P., and Tjahjono, D. 1993. Improving Software Quality through Computer Supported Collaborative Review. *Proceedings of the 3rd European Conference of Computer Supported Cooperative Work,* September 1993.

Johnson, P., Tjahjono, D., Wan, D., and Brewer, R. Experiences with CSRS: An Instrumented Software Review Environment. *Proceedings of the Pacific Northwest Software Quality Conference,* October 1993.

Johnson, P. 1994. Supporting Technology Transfer of Formal Technical Review through a Computer Supported Collaborative Review System. *Proceedings of the 4th International Conference on Software Quality.*

Johnson, P. 1996. Design for Instrumentation: High Quality Measurement of Formal Technical Review. *Software Quality Journal,* Vol. 5, Nr. 1, March 1996.

Johnson, P., and Tjahjono, D. 1997. Assessing Software Review Meetings: A Controlled Experimental Study using CSRS. *Proceedings of the 1997 International Conference of Software Engineering (ICSE-97),* May 1997.

Johnson, P., 1998. Reengineering Inspection. *Communications of the ACM,* Vol. 41, Nr. 5, February 1998.

Johnson, P., and Tjahjono, D. 1998. Does Every Inspection Really need a Meeting" *Empirical Software Engineering,* Vol. 3, Nr. 1, 1998.

Jones, S. 1995 Identification and Use of Guidelines for the Design of Computer Supported Collaborative Writing Tools, *Computer Supported Cooperative Work,* Vol. 3, Nr. 3-4.

Jones, C. 1998. Bad Days for Software. *IEEE Spectrum*, Vol. 35, Nr. 9, September 1998.

Kiesel, N., Schuerr, A., and Westfechtel, B. 1993. GRAS: A Graph-oriented (Software) Engineering Database System". *Information Systems,* Vol. 20, Nr. 11, November 1993.

Knight, J., and Myers, E. 1993. An Improved Inspection Technique. *Communications of the ACM,* Vol. 36, Nr. 11, November 1993.

Lees, B., Jenkins, D. 1995. Supporting Traceability in Conceptual Design. *IEE Colloquium on Design Systems with Uses in Mind: The Role of Cognitive Artifacts",* December 1995.

Leveson, N., Heimdahl, M. P. E., Hildreth, H., and Reese, J. D. 1994. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering,* Vol. 20, Nr. 9, September 1994.

Lougher, R., and Rodden, T. 1993(a) Group Support for the Recording and Sharing of Maintenance Rationale, *Software Engineering Journal,* Vol. 8, Nr. 6, November 1993. (T)

Lougher, R., and Rodden, T. 1993(b) Supporting long-term collaboration in software maintenance. *Conference on Organizational Computing Systems.* ACM. 1993

MacDonald, F., Miller, J. Brooks, A., Roper, M., and Wood, M. 1995. A Review of Tool Support for Software Inspection. *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering,* July 1995.

MacDonald, F., and Miller, J. 1999. A Comparison of Computer Support Systems for Software Inspection. *Automated Software Engineering*, Vol. 6, Nr. 3, March 1999.

MacDonald, F., Miller, J. Brooks, A., Roper, M., and Wood, M. 1996. Applying Inspection to Object-Oriented Code. *Software Testing, Verification, and Reliability,* Vol. 6, Nr. 2, June 1996.

Malone, T., Grant, K., Turbak, F., Brobst, S., and Cohen, M. 1997. Intelligent Information-sharing Systems. *Communications of the ACM,* Vol. 30, Nr. 5, May 1997.

Malone, T., Lai, K-Y., and Fry, C. 1995. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. *ACM Transactions on Information Systems,* Vol. 13, Nr. 2, April 1995.

Mashayekhi, V. 1995. Distribution and Asynchrony in Concurrent Software Engineering. *PhD Thesis, University of Minnesota Department of Computer Science.*

Mashayekhi, V., Drake, J., Tsai, W., and Riedl, J. 1993. Distributed Collaborative Software Inspection", *IEEE Software,* September 1993.

Mashayekhi, V., Feulner, C., and Riedl, J. 1994. CAIS: Collaborative Asynchronous Inspection of Software. *2nd SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

Mashayekhi, V., Glamm, B., and Riedl, J. 1996. AISA: Asynchronous Inspector of Software Artifacts. *University of Minnesota Technical Report TR-96-022*, March 1996.

McAlpine, K., and Golder, P. A New Architecture for a Collaborative Authoring System – Collaborwriter. *Computer Supported Cooperative Work,* Vol. 2, Nr. 3, 1994.

MS Word. http://www.microsoft.com/office/word

Murphy, P., and Miller, J. Asynchronous Software Inspection. *Proceedings of the 8th IEEE International Workshop on Software Technology and Engineering Practice, Incorporating Computer Aided Software Engineering*, 1997.

Neuwirth, C., Moris, J. Regli, S., Chandhok, R., and Wenger, G. 1998. Envisioning Communication: Task-Tailorable Representations of Communication in Asynchronous Work". *Proceedings of ACM 1998 Conference on Computer Supported Collaborative Work (CSCW'98),* November, 1998.

Nyquist, E., and Henricson, N. 1992. Programming in C++, Rules and Recommendations. Ellemtel Telecommunication Systems Laboratories Technical Report, Number M 90 0118 Uen, Box 1505, 125 25 Alvsjö, Sweden.

Object Design, Inc. www.odi.com

Ovsiannikov, I., Arbib, M., and McNeill, T. 1999. Annotation Technology. *International Journal of Human-Computer Studies,* Vol. 50, Nr. 4, April 1999.

Parnas, D., and Weiss, D. 1985. Active Design Reviews: Principles and Practices. *Proceedings of 7th International Conference on Software Engineering (ICSE'85),* August, 1985.

Perpich, J., Perry, D., Porter, A., Votta, L., and Wade, M. 1997. Anytime, Anywhere Code Inspections: Using the Web top Remove Bottlenecks in Large-scale Software Development. *Proceedings of the 19th International Conference on Software Engineering (ICSE'97),* May, 1997.

Pino, J. 1996. A Visual Approach to Versioning for Test Co-authoring. *Interacting with Computers,* Vol. 8, Nr. 4, December 1996.

Pohl, K. 1996. PRO-ART: Enabline Requirements Pre-Traceability. *Proceedings of the 1996 International Conference on Requirements Engineering (ICRE'96).*

Rational Software. Rational Software Corp. www.rational.com

Remark. Infodata Systems Inc, http://www.ambia.com/remark.htm

Rodden, T. 1996. Populating the Application: A Model of Awareness for Cooperative Applications. *Computer Supported Cooperative Work Conference,* 1996.

Smith, T. 1992. READS: A Requirements Engineering Tool. *Proceedings of the IEEE International Symposium on Requirements Engineering (ISRE'92).*

SDT Corp. http://sdtcorp.com

Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. 1988. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM,* Vol. 31, Nr. 11, November 1988.

Stein, M., Heimdahl, M., and Riedl, J. 1998. A General Framework for Interconnecting Annotations of Software Systems. *Proceedings of the 22nd Annual International Computer Software and Applications Conference (COMPSAC'98),* August 1998.

Stein, M., Riedl, J., Harner, S., Mashayekhi, V. 1997. A Case Study of Distributed, Asynchronous Software Inspection *Proceedings of the 1997 International Conference of Software Engineering (ICSE-97),* May 1997.

Streitz, N., Haake, J., Hannermann, J., Lemke, A., Schuler, W., Schutt, H., and Thuring, M. SEPIA: A Cooperative Hypermedia Authoring Environment. *Proceedings of ECHT'92, 2nd European Conference on Hypertext and Hypermedia,* December 1992.

Subramanium, G., and Byrne, E. 1998. Reengineering the Class – An Object-Oriented Maintenance Activity. *Proceedings of the 22nd Annual International Computer Software and Applications Conference (COMPSAC'98),* August 1998.

Takahashi, K., Potts, C., Kuman, V., Ota, K. and Smith, J. 1996. Hypermedia Support for Collaboration in Requirements Analysis. *Proceedings of International Conference on Requirements Engineering (ICRE'96)..*

Tammaro, S., Mosier, J., Goodwin, N., and Spitz, G. 1997. Collaborative Writing is Hard to Support: A Field Study of Collaborative Writing. *Computer Supported Cooperative Work: The Journal of Collaborative Computing.* Vol. 6, Nr. 1, 1997.

Tarr, P., and Clarke, L. 1993. PLEIADES: An Object Management System for Software Engineering Environments. *SIGSOFT Software Engineering Notes*, Vol. 18, Nr. 5, December 1993.

Tervonen, I. 1996. "Support for Quality-Based Design and Inspection. *IEEE Software*, January 1996.

Tervonen, I., Harjumaa, L., and Iisakka, U. 1999. The Web Generation of Software Inspection: A Process with Virtual Meetings and On-line Recording. *to appear in the Ninth Software Technology and Practice (STEP) Conference*, August 1999.

Thompson, C., and Riedl, J. 1995. Collaborative Asynchronous Inspection of Software using Lotus Notes. *University of Minnesota Department of Computer Science Technical Report TR-95-047*, June 1995.

Tjahjono, D. 1995. Comparing the Cost Effectiveness of Group Synchronous Review Method and Individual Asynchronous Review Method using CSRS: Results of Pilot Study. *University of Hawaii Department of Information and Computer Sciences Technical Report ICS-TR-95-07*, January 1995.

Third Voice. www.thirdvoice.com

van der Gerst, T., and Remmers, T. 1994. The Compuer as a Means of Communication for Peer-Review Groups. *Computers and Composition*, 1994.

Visible Corp. www.visible.com

Visual Studio. Microsoft Visual Studio. msdn.microsoft.com/vstudio

Votta, L. 1993. Does Every Inspection Need a Meeting? *Proceedings of the 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November, 1993.

Walsh, J., and Ungson, G. 1991. Organizational Memory. *Academy of Management Review*, Vol. 16, Nr. 1, 1991.

Yee, K-P. 1998. CritLink: Better Hyperlinks for the WWW" *Foresight Institute report*, available at http://crit.org/~ping/ht98.html.

Yourden, E., Glass, R., Booch, G., Jackson, M., Lehman, M., and Andriole, S. A Tale of Two Futures: Software Industry. *IEEE Software*, Vol. 15, Nr. 1, 1998.

APPENDIX 1 – FAULT COLLECTION QUESTIONNAIRE

This questionnaire was given to participants in the experiment described in Chapter 8 after the Fault Collection phase of inspection. The questionnaire was distributed in hard-copy form, with sufficient space left between questions for participants to write their answers. Substantive answers to some of the questions are given after this sheet.

Questionnaire for Fault Collection Phase of Software Inspection Study

Please briefly answer the following questions. We need the identifying questions to know which tool version you used.

Identifying questions:

1. Are you a study participant - did you sign a consent form in class April 3rd ? _______ (if not, you don't need to fill out this questionnaire, although you may if you wish.).
2. Inspection ID: _________ ("flm<n><x>", with <n> = 0-4, and <x> = *a*, *b* or blank)
3. Does your version of AnnoSpec support "Logical Items"? __________________
   (2 Dark blue columns to the left of the 2 aqua columns in the code display)
4. Does your version of AnnoSpec support "Foreign Annotations"?________________
   (Yellow column to the right of the aqua columns in the code display)

Questions on Fault Collection:

1) How much time, to the nearest hour, do you estimate that you spent in the fault collection phase of this software inspection? ____________________
2) What did you find most difficult to use about the AnnoSpec tool, once it was acceptably debugged?
3) What enhancements to the user interface of the AnnoSpec tool would make it easier to use?
4) In your work, do you perform software inspections? _______________
   a) If your answer to #4 was "yes", do you perform them manually or with tool support?
      i) If your answer to #3a was "with tool support", do you use a publicly available tool, or one proprietary to your company?
5) What extra capabilities would help the AnnoSpec tool support fault Collection better?

6) Anything else you'd like to say about the assignment or the tool?

Selected Responses from Questionnaire

Selected questions from above appear in **bold face,** with *italicized responses.* [Bracketed roman type denotes our comments.]

Q: What did you find most difficult to use about the AnnoSpec tool, once it was acceptably debugged?

Speed, number of buttons

Updates were a bit slow sometimes

Slow…

Download time was too long

Too many windows and confirmations.

I had to print out the code because the format was difficult to see the larger aspects of the code.

The button user interface was a bit cumbersome.

It's not a good way to do the initial reading of the code.

Navigation between a component's implementation and its header file is difficult.

Relationship between different classes is not clear.

I didn't like the fact that separate windows were used for the annotations.

Performance wasn't very good. Always going back to the HTTP server was costly when connected via 28.8K or even 56K dial-in.

Navigation could be improved – difficult to locate annotations

Navigating ability and also the response time.

Interface required too many mouse clicks – very cumbersome to add annotations.

Managing the various open windows.

Not seeing an updated view right away of the file once an annotation was made.

Unable to change/delete annotations…

Trying to view the method annotation View and the Annotation form at the same time (when making reference to an annotation).

Q: What enhancements to the user interface of the AnnoSpec tool would make it easier to use?

(Keep) *track of which modules we had already reviewed.*

... more interactive – such as making it into a Java applet

I write code in an editor, why can't I review it in an editor as well?

Ability to see the complete code from the screen without the buttons.

Instead of buttons, use HTML links.

A Java interface in a style closer to a file list/ editor might be easier to use.

HTML frames instead of buttons

I could use some sort of auxiliary checklist in the files view which would allow me to check off which files I had already looked at in this session.

I would like to see the annotations expanded in-line within the actual document/ source under review, or at least within the same window.

Ability to go to the next annotation after you review a comment (and reply to it) without getting back to the code source.

Don't use so many windows.

Provide a link to requirements and detailed design.

Make it faster (transfer less data).

Make an annotation for more than one line at a time.

Requirements traces.

Q: What extra capabilities would help the AnnoSpec tool support Fault Collection better?

A "reminder" checkbox to mark a line of code to look at again later.

I would like to see where a piece of code has been used, such as variables.

Be able to append to an annotation ... to correct an error

Be able to link similar annotations

Select a block of code to comment on

Ability to add types of defects rather than the static list or "other" category.

A toggle between body (.cpp) and spec (.h) files for an object.

On initial file list, show next to each file who has reviewed the file [for the Moderator].

Allow annotations to be expanded inline, or minimally, in the same window as document being reviewed.

Minimize space for headings, buttons, and other tool controls.

Identification of comments by color.

It would be nice to be able to see source code (the whole file) while doing annotation.

The ability to view a list of the locations of the annotations by certain inspectors.

A way to see and/ or set the deadlines and metrics of the inspections.

Provide the design and then the traceability into the code.

Integrate with Visual Studio so you can use its class viewer and search capabilities.

the ability to view different files at the same time. Currently, we can only view one file at a time.

Printing capability

Linking capability that will link from a particular ".cpp" file to its header file and vice versa.

Less graphics, more speed.

Q: Anything else you'd like to say about the assignment or the tool?

It was OK tool but not robust.

Too many files for very limited time.

A better set of code/ design to comment on.

Some type of multiple windowing scheme to track through activities.

Need a better design document.

Even when using this tool I printed out the code. It is easier for met to look at the code on paper and then use the tool to annotate.

Less defects.

...I'd suggest a smaller amount of code as the assignment.

APPENDIX 2 – DISCUSSION QUESTIONNAIRE

This questionnaire was given to participants at the end of the experiment described in Chapter 8. The questionnaire was distributed in hard-copy form, with sufficient space left between questions for participants to write their answers.

Final Questionnaire on Software Inspection Study – TWO-SIDED

What is the your Inspection Number? (0, 1a, 1b, 2a, 2b, 3a, 3b, 4a, 4b)?: _________

Questions for Everyone:

1) How much time, to the nearest hour, do you estimate that you spent in the discussion phase of this software inspection? ____________________
2) What did you find most difficult to use about the AnnoSpec tool in the discussion phase?
3) What would have made it easier to reply to annotations?
4) Do you feel that holding the electronic discussion was a useful addition to inspection? If so, why? If not, what changes to the inspection protocol or the tool could make an electronic discussion such as we had more useful?
5) Overall, has this lab changed your opinion about software inspection? If so, how?

Question for those with Inspections IDs 2[ab] or 4[ab]:

Your version of the tool gave you the ability to annotate not only lines, but also other "items", such as Classes and Methods.

6) Did you find this ability worthwhile?
7) Is so, why? If not, do you think the idea itself is of limited usefulness, or might tool enhancements make it more useful?

Question for those with Inspection IDs 3[ab] or 4[ab]:

Your version of the tool gave you the ability to view "foreign" annotations, that is to view an annotation from a location in the inspection material other than the location at which the annotation was first made.

8) Did you find it useful to be able to view these "foreign" annotations?
9) Is so, why? If not, do you think the idea itself is of limited usefulness, or might tool enhancements make it more useful?
10) Anything else you'd like to say about the assignment or the tool? *(Note: There were no responses to this question.)*

Selected Responses from Questionnaire

The format for these selected responses is the same as in Appendix 1.

Q: What did you find most difficult to use about the AnnoSpec tool in the discussion phase?

too often a button closed the web browser, rather than just exiting the tool.

Very tedious to go through each annotation in each class. It was slow and unavailable Thursday night.

The unreliability of the tool was by far the most difficult thing. I was unable to login Tuesday, Wednesday, or Thursday night. [odd, because the tool was available throughout the Discussion except for the 12-hour crash on the final Thursday].

too slow…

I think the most difficult aspect is getting a response within an acceptable time frame, eventually I began to forget the thought line I had.

Note being able to see all the annotations and replies on one screen. It took a long time to [illegible] *switching between them.*

The discussion threads were not very intuitive, in my humble opinion html does not create a very effective discussion environment.

The speed with which web pages loaded.

When viewing complete set of annotations for a file and then adding an annotation, when I came back, list of annotations was truncated.

Spent too much time switching from viewing the source code to viewing the file index.

The network is far from production quality.

Sometimes didn't work when convenient to use it.

When working through files a <next file> button would be useful. Wouldn't have to go out to the file index them.

Not seeing the source code when getting inside the reply window.

Something that can't be answered right away. Something that can be done within 2 minutes may have to take up to days to get done. Result: the entire code review process becomes to lengthy.

I couldn't find a key to the letters on the annotation buttons. What was the distinction between "v" and "N". [explained in user manual]

Putting issues together that involve different parts of a program.

As a moderator, having to reply to an annotation before marking it as resolved was annoying. Many items were so obvious there should have been a "reply and resolve" option.

Just waiting for the tool to load some of the files...

I tried to access it last night [server crash] *several times. But everytime I clicked on the view annotate button, the timeglass just sits there forever. So I gave up. I think it's the network problem.*

Q: What would have made it easier to reply to annotations?

View all annotations inline with code.

Having a way to view all annotations on one page. Speed of navigation.

Going through and resolving each issue was also very time-consuming, a Moderator could easily spend a good day going through everything.

Add a button to view all annotations and comments for one file at the same time. [such a button exists, and was displayed prominently on the code display].

A reminder-like tool

Less screen flows.

A lot of times annotation in different parts of the code were related. Having the ability to link annotations might be useful (to avoid making the same note in many places).

Better response time.

A short list by subject. Perhaps there could be a list with only one line of the comment that then expands when selected.

It would be nice to be able to go back my previously made annotations and make changes.

Single window for everything.

The form of the web page, i.e. one panel shows source code, one panel allows replying.

The review process has to be changed! All reviewers need to do the review at the same time. Plus the software may need to add something like an on-line chat box where reviewers can talk to each other.

Faster response!

Compress the format so you can see more on screen when you are in discussion.

Faster response time.

Viewing the subject of the initial annotation from the file-view page, without requiring a round trip to the server.

Q: Do you feel that holding the electronic discussion was a useful addition to inspection? If so, why? If not, what changes to the inspection protocol or the tool could make an electronic discussion such as we had more useful?

Yes. Can't always get all people together to discuss something. Keeps a good record of peoples comments that may have been lost otherwise.

No. It is difficult to make discussion. Sometimes I didn't know if the comment was valid or not – so in those cases I didn't respond…

It was useful, but the inspection has a tendency to drag on…

Yes, because we could discuss the points with each other and arrive at some final decisions.

I question if it's worth the time. The collection phase was valuable but I think this discussion phase might be better off-line with the authors resolving the issues with the appropriate people.

Yes, but it would be nice to be able to post a schedule so that when someone logged in they could see the time line…

Yes, it is much more productive to comment in discussions and use the combined knowledge of all participants.

Yes, it allows people to perform inspections without being formally scheduled for meetings.

Yes, better than fault collection phase. Other comments prompted further thinking and identification of faults.

As a moderator, I think the ability to make general announcements to the other team members within the tool would have been a useful motivator.

I felt the electronic discussion worked well because you could respond on your own time.

It is useful especially when the inspectors are located in different geographical locations.

Yes, it seems like a valid method to convey thoughts.

Electronic discussion useful but could easily see the discussion phase stretching out for longer than it should.

Yes, electronic had no time constraint. It also allow time for you to do research[?] *before you make decisions.*

I think that debating points in person would be much more efficient than over the internet.

It was useful to a certain extent.

On-line requirements and a way to link lines of code to them directly.

Yes – gather opinions helps.

If someone agrees with an annotation, they are unlikely to actually add a reply annotation, because of the fact that they have to type in a whole new annotation. they should be able to just click a single button or check a box that they concur.

I like the tool for the inspection, but would still prefer an off-line discussion. This is a good start to a discussion, but resolving is difficult without a face to face discussion.

It's a good idea since we can look at the annotation on the same screen while we are interacting with each other.

Q (groups 2 and 4): Your version of the tool gave you the ability to annotate not only lines, but also other "items" such as Classes and Methods. Did you find the ability worthwhile? If so, why? If not, do you think the idea itself is of limited usefulness, or might tool enhancements make it more useful?

Yes – ability to discuss other classes or methods which may work better.

Yes, when a class and/ or method as a whole has an issue, it is a nice feature to annotate the entire class/method.

Yes, but there was a lot of "repeat annotations". It would be great to be able to link all the annotations together into one button.

Yes. Effective in annotating missing requirements and general errors within classes and methods.

Absolutely. Especially for inspecting OO-based programs. A lot of programming errors occurs during the interface definition phase.

Not really. Not everyone used the same annotation categories the same way. Many times I had to look at both categories before responding. I think it's of limited usefulness, one can always annotate the first line of a class/method if the comment applies to the whole class/method.

Yes. I think for inspections to be useful an ability to have a higher level discussion is important.

No, I think the normal inspection procedure is adequate. I think it is a "nice" to have feature. Not a "need" or "required" feature.

Yes. Sometimes the annotation applied to the functionality rather than a line, and a method annotation indicated that.

Yes. It made the review process easier.

I don't think I used it, but I can certainly see the value.

Yes in a few cases more familiarity with the tool would help. Helps focus discussion on a class or method and not just an instance.

Sometimes, like when pointing out that a method as a whole did not correctly implement a requirement. There is really no good way to point out missing items of *a class (methods, attributes, implemented requirements), other than putting an annotation in the general class "item".*

Yes. Because you can cover several problems within a class or method with one general annotation.

Yes, it really does. I just need to learn more from the experience to appreciate it.

Q(groups 3 and 4): Your version of the tool gave you the ability to view “foreign” annotations, that is to view an annotation from a location in the inspection material other than the location at which the annotation was first made. Did you find it useful to be able to view these “foreign” annotations? If so, why? If not, do you think the idea itself is of limited usefulness, or might tool enhancements make it more useful?

Not really. Idea is of limited utility.

Yes. However, it seems a bit disorienting to view a foreign annotation out of its original context. It needs a bit more visual differentiation from a local annotation.

Yes, very. The idea is very useful because it helps reduce the effort.

Yes, somewhat useful. It is useful to view foreign annotations because a class might use methods from another class and there are useful annotations relevant to these methods.

I found no real use at this time but I may later in practice.

No, actually was confused by it.

Yes, since it provides some reference ability. Make it more robust.

Not really. I think it doesn't hurt. I would have preferred to see links to the requirements document or class hierarchy.

Yes. It made the review process easier.

Yes. They helped bring the discussion to a single thread but it was still difficult…

I found it confusing, because they seemed to be misplaced annotations. Also, I wasn't told what they meant, so I found them useless.

I found it confusing at first but now it makes sense after using the tool for a while.

APPENDIX 3 - INSPECTION SYSTEM FORMALIZATION

Introduction

This appendix shows an alternate formalization of the graph structure used as a basis for the work in Chapter 5 of this dissertation. The formalizataion is expressed in the Z modeling language.

Static Structure of Inspection System

We take a broad view of (annotatable) Items. An Item can be a requirements document, an individual requirement, a design document, a class in the class diagram, or an attribute in a class description. We impose no granularity or other restrction on what can be an Item; that classification is left to the user of an inspection system.

[ATTRIBUTE]

```
∪Item______________________________________________
→ Name : NAME
→ ItemAttributes : Φ ATTRIBUTE
∩______
∠__________________________________________________
```

A Link is a relation that maps an Item to a set of other Items. Examples of Links may be "Inherits From", "Is composed of", "Is called by", etc. There is really no restriction on what relationships between Items are captured with the links.

```
∪Link______________________________________________
→ Map : Item φ Item
→ LinkAttribute : Φ ATTRIBUTE
∩______
∠__________________________________________________
```

The Artifacture is the collection of all Items together with the links that link them together to a graph structure.

```
∪Artifacture_______________________________________
→  Items : Φ Item
→ Links : Φ Link
∩______
→ A m ε Links • (dom m.Map ζ Artifacts) ∧ (ran m.Map ζ Artifacts)
∠__________________________________________________
```

We make a distinction between a production Item *PI* that is an Item used in the production of the software and an annotation Item *AN* that is attached as an annotation to another Item.

→ *PI* : Φ *Item*
→ *AN* : Φ *Item*
∩______
→ *PI* ζ *Artifacture.Items*
→ *AN* χ *Artifacture.Items*
→ *PI* I *AN* = 0
→ *PI* Y *AN* = *Artifacture.Items*

A view is a subset of the Items. Conceptually, a view is what an inspection system may display to the user for viewing. This is typically a subset of the Items selected based on some selection criteria.

→ *View* : Φ *Item*
∩______
→ *View* ζ *Artifacture.Items*

An Item is visible in a view if it is part of the view.

→ *visible* : (*Item* x *View*) □ *bool*
∩______
→ A *a* : *Item*, A *v* : *View* ·
→   *visible*(*a*,*v*) = *a* ε *v*

To simplify the model we introduce a new data type.

Path == seq (*Item* □ *Item*)

A path between two Items *x* and *y* along a relation *r* is a sequence of maplets where the first element of the first maplet is *x*, the second element of the last maplet is *y*, and for each maplet in the sequence the second element of the maplet is the same as the first element of the maplet that follows.

→ *path* : *Item* ξ *Item* ξ *Link* □ *Path*
∩______
→ A *x*, *y* : *Artifacture.Items*, A *l* : *Artifacture.links* ·
→   **let** *p* == *path* (*x*, *y*, *l*) ·
→   A *m* : *squash p* · *m* ε *l.Map* □
→   *x* = *first head p* □
→   *y* = *second last p* □
→   (*length p* > 1 □
→   A *i* : {1, …, *length p* - 1 } · *second p*(*n*) = *first p*(*n*+1))

The distance between two Items along a link is the shortest path between the Items.

```
→ distance : Item ξ Item ξ Link ☐ N
∩______
→ A x, y : Artifacture.Items,  A  l : Artifacture.links ·
→    let P == { p : Path · p = path x y l · p } ·
→      E  a : P, A b : P · length a ≤ length b ☐ distance ( x, y, l ) = length a
```

A route is the set of Items a path goes through.

```
→ route : Path ☐ View
∩______
→ A p : Path ·
→    route (p) = { a : Item · (Em : squash p · a = first m ☐ a = second m ) · a }
```

This is not the most elegant definition. A neighborhood is the set of all Items within a certain distance along some Link.

```
→ neighborhood: Item ξ Link ξ N ☐ View
∩______
→ A a : Artifacture.Items, A l : Artifacture.Links, A i : N
→    neighborhood(a,l,i) ==
→      Y { p : Path  ·  Ay : Item ·
→        p = path a y l ☐ distance a y l ≤ i · route p }
```

We extend the definition of a neighborhood to apply to a set of Items.

```
→ Neighborhood: Φ Item ξ Link ξ N ☐ View
∩______
→   Neighborhood(A, l, i) = Y { a ε A  · n = neighborhood a l i · n }
```

There is a link in the artifacture that represents the annotation links.

```
→ anno_link : Link
∩______
→   "annotationLink" ε anno_link.Attribute
```

A native annotation is connected directly to the production Item with an annotation link.

```
→ native_annotation : Item ξ Item ☐ bool
∩______
→ Ap ε PI, A a ε AN  ·
→    native_annotation(p,a) = El : anno_link · ( p ☐ a ) ε l.Map
```

A foreign annotation is not connected directly to the production Item, but there is a path along a certain link to some other Item to which this annotation is connected with an annotation link.

```
→ foreign_annotation : Item ξ Item ξ Link ☐ bool
∩______
```

→ A *p* ε *PI,* A *a* ε *AN,* A *l* ε *Artifacture.Links* ·
→ *foreign_annotation(n,a,l)* = ! *native_annotation n a* □
→ E*x : Annotation* · *native_annotation x a* □
→ E*p : Path* · *p* = *path n x l*

Filtering the Annotations

A filter is simply a mapping from a view to another view. The notion of filters will be defined as "generic" schemas that can be instantiated to specific filters. In general, a filter is a mapping from one view to another.

→ *filter : View*□ *View*

Narrowing Filters

A narrowing filter removes some Items from a view. The general structure of a filter is shown below. A filter removes Items from a view based on some arbitrary selection predicate *PREDICATE* : *Item* ϕ *bool*. We call a narrowing filter with the selection predicate *q* an *annotation filter* if A*a* : *Item* · *q a* □ *a* ε *AN*. That is, the resulting view after applying an annotation filter consists solely of annotations.

∪**narrow_filter[PREDICATE]**___________________________________
→ *vu?* : *View*
→ *new_vu!* : *View*
∩______
→ *new_vu!* = { *x* ε *vu?* • *PREDICATE x* • *x* }
∠____

Broadening Filters

A broadening filter expands the view based on some expansion criteria expressed over the artifacture.

The broadening filter will be parameterized with the following parameters: *BROAD_REL* : *Link*, *SEL_PRED* : *Annotation* ϕ *bool* and *SIZE* : N. In the definition below, *SEL_PRED* selects the Items we are interested in "expanding" and *BROAD_REL* is the link along which we want to perform an expansion. *SIZE* is how much of an expansion we want to make, that is, how many steps do we want to follow the link.

∪**broadening_filter[BROAD_REL,SEL_PRED,SIZE]**_______________________
→ Ξ*Artifacture*
→ *vu?* : *View*
→ *new_vu!* : *View*
∩______
→ *new_vu!* = Y { *x* ε *vu?* • *SEL_PRED x* • *neighborhood x BROAD_REL SIZE* }

∠____

Filtering Example

To illustrate the use of filters, consider an artifacture where the Items represent the various entities in a UML class diagram (for instance, classes, attributes, relations, operations, etc.) and the links represent the relationships in the class diagraion (for instance, inherits from, is composed of, is attribute of, etc.). We assume that certain properties of the Items and links are recorded in the "Attributes" portion of links and Items. Properties recorded may include the author, the time the Item (or link) was created, etc.

Assume we have the link *inheritsFrom* that maps an Item representing a class to a set of Items representing the children of this class. We make the relation in the link total by mapping all Items without children to the empty set. Also assume that we have the link *annotates* which represents that an Item annotates another Item. Here we also make the link relation total by mapping any Item without an annotation to the empty set.

Now, assume we also have defined four predicates *authorMike, isAnnotation, selectAll,* and *isSelected* mapping a view to Boolean.

→ *authorMike* : *Item* ϕ *bool*
∩______
→ A*a* ε *Artifacture.Item* ·
→ *authorMike* (*a*) = *Author_Mike* ε *a.ItemAttributes*

→ *isAnnotation* : *Item* ϕ *bool*
∩______
→ A*a* ε *Artifacture.Item* ·
→ *isAnnotation* (*a*) = *Annotation* ε *a.ItemAttributes*

→ *selectAll* : *Item* ϕ *bool*
∩______
→ A*a* ε *Artifacture.Item* ·
→ *selectAll* (*a*) = *true*

→ *isSelected* : *Item* ϕ *bool*
∩______
→ A*a* ε *Artifacture.Item* ·
→ *isSelected* (*a*) = *isSelected* ε *a.ItemAttributes*

With these definitions, we can define a filter that will give us all annotations on the children Mike has created of the classes we have selected in a view. In a realization of this inspection formalism, we can envision this being done by simply selecting the classes we want to "expand" (by "shift click" for example) and selecting the filter from a drop down menu. Also, assume we are interested in looking only at annotations of the immediate descendants of the selected classes (at a distance 1 from the class along the inherits from link).

inheritsExpansion ☐ *narrow_filter* [*isAnnotation*] ;
broadening_filter [*annotations*, *selectAll*, ∞] ;
narrow_filter [*authorMike*] ;
broadening_filter [*inheritsFrom*, *isSelected*, 1]