A General Framework for Interconnecting Annotations of Software Systems

Michael V. Stein
Mats P. E. Heimdahl
John T. Riedl
Department of Computer Science and Engineering
University of Minnesota
200 Union St. SE
Minneapolis, MN 55455 USA
+1 612 626 8396 +1 612 625 2068 +1 612 624 7372
mstein@cs.umn.edu heimdahl@cs.umn.edu riedl@cs.umn.edu

Abstract

Computer-supported annotation of software systems and their documentation, including design documentation and source code, is a common and important software engineering activity. Annotated documentation is used in both formal software inspection and informal software maintenance. Viewers of annotated systems may understand the software more easily if annotations are visible not just from the annotated item itself, but from other, related items. We propose a general framework for interconnecting annotatable items in software systems to achieve this visibility. We describe filtering and broadening rules that viewers can use to select the annotations they desire to see. We illustrate this framework in the context of object-oriented software system development.

Keywords

Annotation, information filtering, object-oriented software development.

1. ANNOTATION OF SOFTWARE SYSTEMS

Annotation is a significant component of team collaboration in a wide variety of settings[5]. One such setting is software engineering. Annotation of textual or graphical documents pertaining to software systems is a common and important software engineering activity. Computerized development tools incorporating annotation have become available in recent years. They are used in diverse areas such as annotating source code to explain design rationale[8] and performing formal software inspection of a wide range of systems[9][4].

Systems annotated with present-day tools suffer from two problems. 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 documentation.

This paper presents a general framework for enhancing annotation of software systems, by allowing users to customize the set of annotations that they see when viewing an annotated document. We argue that such customization may help users enhance their understanding of the system. We then describe the theoretical model underlying our annotation framework.

1.1. Identifying Interesting Annotations

Viewers of annotated documents wish to see the annotations that others have made on those documents. The default behavior in many annotation tools is to assume that people viewing a document defining the user’s present view of the system are interested in exactly the entire set of annotations that have been made on the annotatable items – the Units of Annotation (UAs) - within that view (e.g., [4][11]). For instance, a viewer of a specific file would see exactly all the annotations made to that file. However, the clutter and delocalization problems may arise when using this default annotation scheme. This section describes how the clutter and delocalization problems arise within annotated documents. Section 3 provides a model to address the above problems, and gives an example of using that model to flexibly annotate software system documentation.

1.1.1. The Clutter Problem. It is well known that a highly annotated document, whether in paper or electronic format, may become cluttered with so many annotations that those annotations interfere with comprehension[1]. To solve this clutter problem, some annotation tools employ filtering – showing only a subset of the annotations of an item[1].
A simple example of filtering is to show only the annotations made by a subset of all annotators of the document. Another example of filtering concerns checklist-based inspection of software[2]. In this case, viewers of a document may choose to filter annotations by the checklist items to which they refer.

1.1.2. The Delocalization Problem. The classic delocalization (or delocalized plan) problem occurs when “pieces of code that are conceptually related are physically located in non-contiguous parts of the program”[14]. A unit of annotation X may have relationships to other, delocalized UAs, and some annotations on those delocalized UAs may be of interest to viewers of X. No annotation tool today allows a user to easily see annotations on these related UAs. To address this problem we propose applying broadening of annotations: showing the viewer of a UA not only annotations on that UA itself, but also annotations on related UAs – essentially, showing the viewer a superset of the annotations on a UA.

The delocalization problem shows up in two ways. We argue that broadening can ameliorate the delocalization problem in both cases.

First, there is the classical delocalization among parts of the system at the same level of abstraction. For instance, a viewer may need to understand one piece of code before being able to understand a different piece. As another example, we have observed in field studies of software inspection that when people recognize an error in the interface between two items A and B of a software system, some report the error on A, and others report the error on B[15]. A viewer of the annotated documentation could benefit from seeing annotations on both A and B.

This delocalization problem is especially severe for object-oriented (OO) software[10]. 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 polymorphism (it is not clear from the static code structure what function will actually run) and parameterization of classes. As a simple example, an annotation of a class C in an OO design may be of interest to those viewing C’s derived classes or base classes.

1.2. Simultaneous Filtering and Broadening

We propose a simultaneous solution to the clutter and delocalization problems. Our solution is to allow viewers interested in a certain UA X to see a subset of annotations on a set of UAs including both X and other UAs related to X. Essentially, they see a subset of a superset of the annotations on the UAs in their view of the document.

When viewers see an annotation A on a UA outside their present view as part of this superset, they do not see the view from which A was made. However, they must be able to navigate to that view easily to better understand the context of the annotation.

The 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.

This paper provides a framework to simultaneously address the clutter and delocalization problems in the domain of OO software development.

1.3. Requirements for a Framework to Interconnect Annotations

The purpose of interconnecting annotations is to enhance annotation activities. To do this, we set out to develop an annotation framework that will enable users to achieve the following goals:

1. View annotated documents, maintaining the functionality of existing inspection/annotation tools.
2. View only a subset of the annotations of visible documents.
3. View annotations on UAs related to a UA being viewed, using relationships among the UAs to decide which other annotations to view.
4. Combine requirements #2 and #3 to view a subset of the annotations on visible UAs, and on UAs related to visible UAs. This requirement, a simultaneous application of filtering and broadening, is the core of our framework.

Requirements 1 and 2 are met by many annotation tools today. Requirement 3, and its extension requirement 4, are not implemented by any tools of which we are aware.

1.4. Realizing a Framework for Interconnected Annotations

The AnnoSpec tool, currently under development, will be a realization of the filtering and broadening requirements described above. AnnoSpec is being built using an OO development model. As such, it can be used to illustrate the ideas of filtering and
broadening through use of the graph-theoretic model we will present later. Pertinent classes within the AnnoSpec system are outlined below.

Figure 1 shows the high-level structure of a portion of the AnnoSpec design in Unified Modeling Language (UML) format.

1.4.1. Descriptions of the Classes. What follows are brief descriptions of the AnnoSpec classes shown in Figure 1. The classes are shown with some annotations. Annotations are identified by the triple (annotation number, author, type). Types are defined in section 3.2.

PhysicalUA (Figure 2) is a virtual class that encompasses all UAs of a document. PhysicalUA has two concrete subclasses, PhysFile and PhysLine. Figure 2 shows the PhysicalUA class with an elided set of its attributes and methods, and with placeholders indicating lines within the method implementations.

PhysFile (Figure 3) represents objects that are source code files. These are annotatable because an annotation can have the scope of an entire file.

PhysLine (Figure 4) represents objects that are lines of a PhysFile. A PhysLine represents one line of a source code file, and is the smallest annotatable unit in the AnnoSpec system.

Annotation (Figure 5) represents an annotation. Annotations can be threaded together, with replies to the original annotation, in a structure reminiscent of threaded internet newsgroups.

Comment (Figure 6) is a virtual class representing a single comment within an Annotation.

BaseAnno (Figure 7) represents a Comment that is a Base Annotation – a comment made directly on software under review.

Reply (Figure 8) represents a Comment that modifies an existing comment.

2. RELATED WORK

Related work falls into three general areas: collaborative authoring tools, software inspection systems, and software development tools.

2.1. 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 necessarily 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[1]. This elegantly supports filtering on one dimension.

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[16].

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"[5]. 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[13]. 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[12]. Users are able to perform type-specific annotation behavior, including changing the scope of some types of annotation.

2.2. Software Inspection Systems

Automated software inspection systems make extensive use of annotation capabilities. MacDonald describes many such systems in detail[9]. We make special note of InspeQ[7], because it makes extensive use of annotations that are typed by checklists. Also, our ideas have been influenced by experiences with our own software inspection tools CAIS[11] and AISA[15].

2.3. Software Development Tools

Software development tools also relate to our framework.

Modern software development tools allow developers to see multiple views of source code, not just the textual view they write[3]. These tools do not allow human annotation to the views, however.

Lougher and Rodden have developed an annotation tool for describing maintenance rationale in poorly-documented legacy systems[8]. This tool supports flexible typing and scoping of annotations.

GRAS[6] and PLEIADES[17] are object
management systems that store OO artifacts in ways that make use of their relationships to other objects. They do not deal specifically with annotations or their information filtering implications. But annotating objects in a such a system would be one way to implement our interconnected annotation model.

3. AN INTERCONNECTED ANNOTATION MODEL

This section describes the graph-theoretic model that underlies our framework for addressing annotation problems, and illustrates how this model would apply to an OO development system.

3.1 Graph-Theoretic Model

**Definition:** The documentation of a software system (or just documentation) is a directed graph $G = [N, R]$, where $N$ denotes the set of nodes of the graph and $R$ denotes the set of edges of the graph connecting the nodes. The set of nodes $N$ can be partitioned into two equivalence classes: (1) The set of UAs, or annotatable nodes, denoted $N_U$, and (2) The set of annotations, denoted $N_A$. Thus, $N = N_U \cup N_A$.

The nature of the nodes is dependent on the nature of the software system. For instance, in OO source code, annotatable nodes could include classes, attributes and methods of those classes, and comments.

The edges $R$ of the graph form the relationships between the nodes. An edge connects an annotation node in $N_A$ to exactly one UA in $N_U$. However, it can be attached to the same UA by multiple relationship edges of different types. That is, one annotation can serve two or more purposes (e.g., identify an incorrect comment embedded in source code, and also indicate the comment contains a misspelling).

Edges also connect two UAs, or two annotations (such as a question about a UA and its response).

**Definition:** A view $V$ of $G$ is a subset of its nodes: $V \subseteq N$. The model does not restrict the set of nodes in a view.

In practice, the set of UAs in a view will be defined in a natural way. Examples of typical views include a source code file, a section of a requirements document, or a single UML class diagram. If $V'$ is a view of the UAs of a document, ($V \subseteq N_U$), then the “traditional” annotated view $V''$ of that same document would be $V'' = V' \cup \{ n \in N_A | m \in V' \text{ and } n \text{ annotates } m \}$.

**Definitions:** The concepts of a path between two nodes, the length of such a path, and the distance between two nodes are defined typically. Within our notation, they are expressed as follows.

A path $P$ between two nodes $n_i, n_k \in N$ is a set of nodes $n_i$ that lead from $n_i$ to $n_k$. Formally, $P(n_i, n_k) = \{ n_i \mid i = 1, \ldots, k \}$. The length of a path $P$, denoted $|| P ||$, is the cardinality of that set of nodes making up the path.

The distance $D$ between the nodes is the length of the shortest path between them. If $P_i$ denotes the set of all paths between nodes $n_j \in N$, then $D(n_j, n_k) = \min_i || P_i(n_j, n_k) ||$.

**Definition:** The neighborhood $s(n)$ of node $n$ is the set of all nodes within a distance $i$ of node $n$. Thus $s(n) = \{ m \in N | D(n,m) \leq i \}$. The neighborhood $S_i(X)$ of a set of nodes $X \subseteq N$ is the set of all nodes that are within a distance $i$ of any node in $X$. Thus $S_i(X) = \cup s(n), \forall n \in X$. We define $s_0(n) = N$.

**Definition:** The type of an edge classifies the edge by various properties. An edge can have a set of such properties, and the set of all edge types is the cross-product of all the possible combinations of the properties.

For instance, edges relating an annotation to a UA may have types of properties such as the author of the annotation, or the type of annotation it is (e.g., misspelling, performance problem). An edge relating two UAs may have properties containing information about its head and tail nodes (e.g., an edge relating a class to one of its attributes).

**Definition:** A restricted path $P_i$ between two nodes $n_i, n_k \in N$ is a set of nodes $n_i$ that lead from $n_i$ to $n_k$, along edges of type $t$. Formally, $P_i(n_i, n_k) = \{ n_i \mid i = 1, \ldots, k \} \text{ and } x_{i+1}$ and $x_{i+1}$ are connected by an edge $r \in R$, of type $t$. $J$.

**Definition:** The restricted distance $D_i$ between two nodes is the length of the shortest path between them along edges of type $t$. Let $P_i$ denote the set of all restricted paths between nodes $n_j, n_k \in N$. Then $D_i(n_j, n_k) = \min_i || P_i(n_j, n_k) ||$.

**Definition:** The restricted neighborhood $s_i(n)$ of a node $n$ is the set of all nodes of $N$ that are within a restricted distance $i$ of node $n$, using edges of type $t$. Thus $s_i(n) = \{ m \in N | D_i(n,m) \leq i \}$. The restricted neighborhood $S_i(X)$ is the set of all nodes that are within a restricted distance $i$ of any node in $X$, using edges of type $t$. Thus $S_i(X) = \cup s_i(n), \forall n \in X$.

Neighborhoods, restricted or unrestricted, can be composed to yield further sets of nodes. For instance, let $r$ be the relationship type “an annotation made by John”, and let $u$ be the relationship type “is a subclass of” (connecting two nodes of type class). Then the following composition of restricted neighborhoods is the set $J$ of annotations made by John on a class $c$, or any of its subclasses: $J = S'_{i_f}(S'_{u}(c))$.

**Definition:** The visibility function $f$ maps the
neighborhood of nodes in one view into another view.

For instance, let $\Sigma$ be a set of restricted or unrestricted neighborhoods around a set of nodes in $N_U$ (UAs) that compose a view $V'$ of the annotatable nodes ($V' \subseteq N_U$). Let $V'$ ($V' \subseteq N$) denote the annotated graph. Then $f$ maps the neighborhoods of the annotatable nodes to the nodes of the annotated graph. That is, $f : (V', \Sigma) \rightarrow V'$. $V'$ contains the UAs of $V'$ plus some annotations, which may be a subset of the annotations of the nodes of $V'$, a superset of those annotations, or some composition thereof.

The core of our framework is to allow $V'$ to contain a subset of a superset of the annotations to the annotatable nodes in $V'$. For instance, someone looking at the implementation of a class could see annotations on the design diagram (for that class (the design view of the class is not in the view that person is looking at, so these annotations would form a superset of $V'$)), or annotations only made by John on UAs in the view (a subset of $V'$), or only annotations made by John on the design or implementation of the class (neither a subset nor a superset of $V'$).

### 3.2. A Graph-Theoretic Model Application

The next two sections apply the graph-theoretic model of section 3.1 to the set of annotated files of section 1.4 as an illustration of filtering and broadening to solve clutter and delocalization problems.

The views of the document are the various files containing classes (Figures 2 through 8).

In this problem, the nodes of the graph (i.e., the UAs of the document) consist of the following:
- Each class.
- Each attribute of each class.
- Each method within each class.
- Each physical line within each method.

The relationship edges of the unannotated graph are listed in the following table. Two relationships require further explanation.

$M$ relates a method $m$ in a class $c$ to concrete methods $m^*$ in subclasses $c^*$ of $c$. & relates a node to its annotation(s). This node may itself be an annotation, in which case & & relate comments in a threaded discussion.

In the annotations, the authors are named “Ann”, “Joe”, “Lee”, and “Rao”. Each author forms a type of annotation node: $\text{a}(a)$, $\text{a}(j)$, $\text{a}(l)$, or $\text{a}(r)$. Additionally, each annotation has a type of its own. For illustration purposes these are $\text{t}(i)$ for an interface comment, $\text{t}(f)$ for a functionality comment, and $\text{t}(g)$ for a general comment.

<table>
<thead>
<tr>
<th>Name</th>
<th>From</th>
<th>To</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>class</td>
<td>superclass</td>
<td>1-many</td>
</tr>
<tr>
<td>$C'$</td>
<td>class</td>
<td>subclass</td>
<td>1-many</td>
</tr>
<tr>
<td>$A'$</td>
<td>attribute</td>
<td>class</td>
<td>1-many</td>
</tr>
<tr>
<td>$M'$</td>
<td>method</td>
<td>class</td>
<td>1-1</td>
</tr>
<tr>
<td>$M^*$</td>
<td>method</td>
<td>concrete meth.</td>
<td>1-many</td>
</tr>
<tr>
<td>$M^{**}$</td>
<td>method</td>
<td>virtual method</td>
<td>1-1</td>
</tr>
<tr>
<td>$L$</td>
<td>method</td>
<td>line</td>
<td>1-many</td>
</tr>
<tr>
<td>$L'$</td>
<td>line</td>
<td>method</td>
<td>1-1</td>
</tr>
<tr>
<td>&amp;</td>
<td>node</td>
<td>annotation</td>
<td>1-many</td>
</tr>
<tr>
<td>&amp;'</td>
<td>annotation</td>
<td>anno'ed item</td>
<td>1-1</td>
</tr>
</tbody>
</table>

Table 1: Relationships within sample document.

### 3.3. Views of the Annotated Document

With the above definitions, it is easy for a user to define a set of annotations of interest. The following compositions, in the spirit of database queries, allow users to see exactly those annotations they desire. After each composition, we list (by number) the annotations that the viewer would see on figures 2-8.

In these examples, we denote the neighborhood $\eta$ of a class $c$, its attributes and methods (including all physical lines within them) by neighborhood $\eta(c) = c \cup S^A(c) \cup S^M(c) \cup S^L(c)$.

Thus, a viewer of PhysicalUA can see all annotations on the class and all its components by defining the neighborhood $S^d(\eta(\text{PhysicalUA}))$. These are annotations 1, 8, and 9 on the figures.

A viewer wanting to see all threaded comments of the above annotations would define $S^d,\eta$.

A viewer wanting to see all annotations made by Joe on all UAs (but not Joe’s replies to annotations made by others) is interested in the neighborhood $S^d,\eta(N_U)$. These are annotations 1 and 4.

A viewer wanting to see all interface comments on class PhysFile is interested in the neighborhood $S^d,\eta(\text{PhysFile})$. These are annotations 5 and 10.

Let $\Theta$ be the set of all methods in class Reply. A viewer wishing to see all annotations on any method of class Reply or on any virtual methods implemented in that class would define the relationship $S^d,\eta(\Theta \cup S^M(\Theta))$. There are no such annotations in the figures.

A viewer wishing to see all annotations on class PhysFile and its superclass (PhysicalUA) is interested in the neighborhood $S^C,\eta(\text{PhysFile})$ and see annotations 1, 5, 7, 8, 9, and 10.
class PhysicalUA {  
  String lastChg;
  Vector annoVector;
  String fName;
  
  PhysicalUA( . . . );
  
  Enumeration getAnnos() {  
    . . .
  }
  
  String getLastChg() {return lastChg;}
  String updLastChange() {  
    . . .
  }
  
  String getFileName() {return fName;}
}

Figure 2: PhysicalUA class, elided and annotated.

class PhysFile extends PhysicalUA {  
  Vector lines;
  
  PhysFile( String n, Inspection i )
  
  Enumeration getLines() {  
    . . .
  }
  
  PhysLine getALine() {  
    . . .
  }
  
  void addLine( int nr, String s ) {  
    . . .
  }
}

Figure 3: PhysFile class, elided and annotated.

class Annotation {  
  String lastChg;
  int number;
  BaseAnno base;
  
  Annotation( . . . ) {  
    . . .
  }
  
  int getNumber() { return number; }
  int getLastChg() { return lastChg; }
  BaseAnno getBase() { return base; }
}

Figure 4: PhysLine class, elided and annotated.

class Annotation {  
  String lastChg;
  int number;
  BaseAnno base;
  
  Annotation( . . . ) {  
    . . .
  }
  
  int getNumber() { return number; }
  int getLastChg() { return lastChg; }
  BaseAnno getBase() { return base; }
}

Figure 5: Annotation class, elided and annotated.

class Comment {  
  String title;
  User author;
  String info;
  String date;
  Vector replies;
  
  Comment( String title, . . . ) {  
    . . .
  }
  
  String getTitle() { return title; }
  String getContents() { return info; }
  . . .
}

Figure 6: Comment class, elided and annotated.

class BaseAnno extends Comment {  
  Reply getAReply( int n ) {  
    . . .
  }
}

Figure 7: BaseAnno class, elided and annotated.

class Reply extends Comment {  
  Reply getAReply( int n ) {  
    . . .
  }
}

Figure 8: Reply class, elided and annotated.
4. CONCLUSION

We identify two problems with modern annotation tools. The first problem is clutter: it is difficult to see interesting annotations when too many annotations are visible at once. The second problem is delocalization: interesting annotations may be attached to parts of the document invisible to a viewer.

We propose a flexible model of annotation that allows users to see annotations on items related to a viewed item, in addition to annotations on the viewed item itself. Users can apply broadening to expand annotation visibility along various relationships defined among annotatable items of a system, and can choose the distance along such relationships to travel when expanding their view. Concurrently, users may apply filtering to see only a subset of annotations of a given item if they so choose, to avoid information overload. They perform filtering by enabling annotation visibility only along certain defined relationships. The power of this model comes from the ability of users to simultaneously expand and contract annotation visibility through the use of these filtering and broadening relationships.

4.1. Future Work

Future work involves extending this annotation model beyond OO documentation to incorporate annotations on versions of a document, and on documents at different conceptual levels.

Broadening can be extended to encompass different versions of the same document. An extension of our model could map the annotations made on one version to future versions in a reasonable manner. Broadening can also interconnect annotations among the requirements, design, and implementation of a software system, aiding traceability.

REFERENCES


