A Conceptual Model for Folding

Technical Report: GT-CS-08-09

October 22, 2008

Spencer Rugaber
College of Computing
Georgia Inst. of Technology
spencer@cc.gatech.edu

Naren Chainani
College of Computing
Georgia Inst. of Technology
nchainani3@mail.gatech.edu

Ogechi Nnadi
College of Computing
Georgia Inst. of Technology
ogechi.nnadi@gatech.edu

Kurt Stirewalt
Computer Science & Engin.
Michigan State University
stire@cse.msu.edu

Abstract

Modern tools for viewing source code typically include a feature that enables readers to fold or unfold selected program segments in support of abstraction. Existing implementations support folding structures that conform to the abstract syntax of the programming language; that is folding by collapsing a structured construct, such as a method, into a single abstract statement. To support other forms of abstraction requires the ability to define folding structures that crosscut the syntactic structure of the language. This paper contributes a conceptual model that generalizes folding to describe both the current state of the practice and also folds involving non-contiguous segments of code. The model characterizes the technical obstacles inherent to supporting non-contiguous folds and describes a variety of policies that can be applied to resolve ambiguity. We also describe a tool called TAJ that supports this capability.

1. Introduction

The primary weapon in program understanding is *abstraction*, the ability to selectively recognize, label and then ignore details. Modern Interactive development environments (IDEs), such as Eclipse and Visual Studio, support abstraction via *folding*, *i.e.*, the ability to selectively hide and reveal source code segments. The folded area is removed from the display, leaving its first line visible, usually accompanied by a user interface affordance, such as a triangle or plus sign. Activating the affordance alternatively expands and collapses the code segment. The affordances are called *holophrasts*, and we call widgets that support them *holophrastic displays*.

In IDEs, the folded segments correspond to syntactic units, such as methods or classes, and the folds must be strictly nested and the folded lines contiguous. However, the abstractions expressing the design decisions present in a program need not be so constrained. For example, a code segment may play a role in multiple, interleaved abstractions, or a given abstraction may be realized in a delocalized fashion at several sites in the code. Hence, program understanding tools could benefit from more powerful folding mechanisms.

The contributions presented in this paper, include the following:

- A conceptual model capable of describing traditional folding
- An extended model that supports delocalized abstractions
- A tool called TAJ that supports the extended model
- A speculative analysis of further extensions in support of program understanding

2. Background

2.1 Holophrastic Display

The term *holophrast* was coined by Wilfred Hansen [8] to denote a visual indicator that stands in place of a more detailed description. Holophrasts correspond in a natural way to depict *chunks*—cognitive abstractions used to abbreviate complex concepts [20]. As such, holophrastic displays support a program reader's need to identify units of meaning in code.

In the late 1970s and early 1980s researchers promoted the idea of *syntax editors*, such as the Cornell Program Synthesizer [24], in which the program construction process was supported by enabling the programmer to instantiate a control structure template after selecting it from a palette. Details could be filled into the template, thereby reducing the overall programming effort and guaranteeing that a program's control structures remained syntactically valid. Once filled in, a control structure and its contents could be hidden (*collapsed* or *folded*) and revealed (*expanded* or *unfolded*) holophrastically.

In modern IDEs, holophrastic display is commonplace. In the Eclipse Java Development Tools (JDT), for example, a small, circled plus sign in the gutter to the left of a source code line indicates that the code for a corresponding method or class is currently hidden but can be made visible by clicking the circle. This situation is depicted in Figure 1. A similar mechanism is available in Visual Studio.

Holophrastic display is not restricted to IDEs. So-called *folded editors* and *outliners* allow the user to select arbitrary contiguous lines of text and then issue a fold

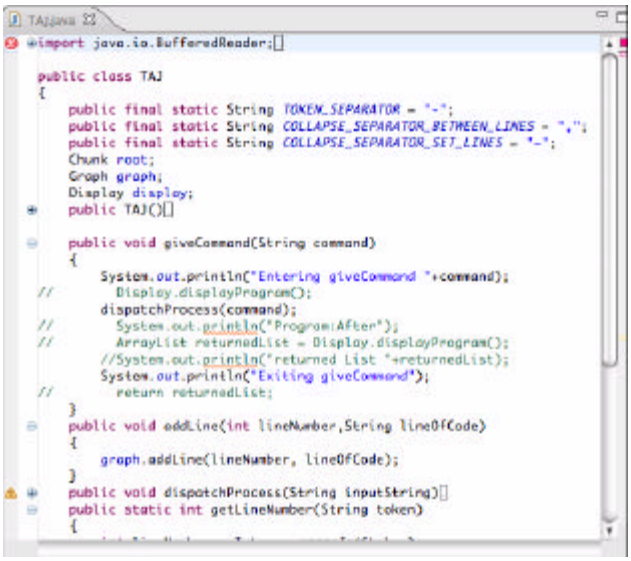

Figure 1. Eclipse source code view with holophrasts

request, thereby hiding the selected lines while leaving a holophrastic visual indicator.

Unfortunately, in these tools, holophrasts are limited in what they can denote. For example, in JDT, only classes, members, comments and import statements can be folded. The C# programming language goes somewhat further, supporting user-defined collapsible segments with the `#region` preprocessor directive. Tools, such as Visual Studio, that recognize this annotation can then collapse arbitrary contiguous lines of code and nested user-specified chunks.

In the above examples, although chunks can be nested, the selected area is restricted to be strictly contiguous. For programs, however, it is sometimes desirable to be able to select non-contiguous areas.

2.2 Non-Contiguous Program Chunks

Current folding technology presupposes a coherent, hierarchical structuring to programs, and early software design approaches such as Stepwise Refinement [26] and Structured Design [22] encouraged this view. However, several factors raise questions about the validity of this assumption. Among the factors are the entropic effects of software maintenance activities, the advent of object-oriented programming languages, program optimizations that encourage the interleaved implementations of disparate program features and Aspect-Oriented Programming (AOP).

Systematic observations by Belady and Lehman [2] pointed out that any *a priori* coherence to the design of a software system tends to dissipate as the software is maintained and enhanced. In particular, they noted that the frequency of occurrence of inter-module dependencies increases as software undergoes maintenance. That is, an enhancement or bug fix requires changes distributed throughout the code base.

Another factor decreasing the locality of software was the advent of object-oriented (OO) programming languages in which actions affecting the state of an object are distributed among its methods, and those methods themselves may be distributed throughout a class hierarchy. The delocalized nature of OO programs can negatively impact understandability; for examples, see [23] with respect to debugging and [5] for code inspections.

In addition to problems inherent in the programming language approach taken, non-contiguity[1] is also a direct result of intentional design practices. For example, [19] points out how subprograms in a (non-OO) library of numeric software frequently produce more than one result. That is, usually in order to avoid duplicating a common segment, the code in a subprogram may interleave several computations around a common, core set of statements. The authors call this phenomenon *interleaving*, and the natural implication is that the implementation of any one design abstraction is spread out in the code in order to allow room for the code implementing other abstractions.

The most direct statement of the problem was first made by Soloway *et al.* [21] in which they describe the problems inherent with delocalized *plans* (design abstractions) and suggest documentation strategies to address the problems. In particular, they suggest a *multi-linked strategy* in which explicit mention is made of delocalized dependencies. They also suggest using literate programming [12] and multiple concurrent views as ways of addressing this problem.

Kiczales *et al.* [10] made delocalization a first-class design consideration with their exploration of AOP. An *aspect* is a module written in an aspect-oriented programming language that expresses a cross-cutting concern. The process of compiling the aspects weaves the aspectual code into the main body of the program at certain preselected *join points*. IDEs that support AOP, such as some of the Eclipse plug-ins described in the next subsection, provide ways to visualize the main code, the aspects and the dependencies among them using multiple window and iconic affordances.

2.3 More Recent Related Work

Several, more recent, studies have explored folding. Mössenböck and Koskimies [15] generalize hypertext links into active text—text with embedded elements such as folds, pictures, bookmarks, annotations, timestamps, widgets and executable scripts. Although the folds must be

1. Terms such as *delocalized*, *scattered*, *tangled* and *cross-cutting* have also been used to describe this phenomenon.

strictly nested, other features such as cross-document links are more general than TAJ, the tool we have developed to support the ideas described in this paper.

Perhaps the work most closely related to ours is the GUPRO IDE developed by Kullbach and Riediger that supports the holophrastic display of preprocessed C programs [13]. The C preprocessor is capable of distorting the original source code via `#define` macros and possibly nested file inclusions. GUPRO is capable of folding and unfolding macro expansions and file includes. Moreover, they present a description of the data structures and algorithms used by the implementation. Their approach supports selections in units of lines and columns, where ours is currently limited to lines only. Their folds can span files boundaries as well, while ours currently is limited to a single file. However, unlike TAJ, their folds are strictly contiguous.

Literate Programming [12] is a software documentation scheme in which descriptive commentary is interleaved with source code. Tools are provided to extract the code for compilation or format the program and its documentation for printing. The embedded documentation can be thought of as folds that can be hidden or displayed. Knasmüsseler extended this idea with concepts from hypertext, where the reader can non-sequentially browse the code by following links [11]. Folds can encompass chunks comprising either code or commentary; however, the folds must be strictly hierarchically organized. It is worth noting that modern IDEs that support folding also support navigation between references and definitions in a hypertext like fashion.

Robillard and Murphy have developed a graph model (Concern Graphs) to describe how delocalized concerns may be documented and viewed [16]. A *concern* is a "conceptual unit of source code of interest to a stakeholder" [17]. A concern graph consists of nodes corresponding to classes, fields or methods and arcs expressing various relations such as calls, reads, writes, checks, creates, declares and super-class. That is, TAJ nodes can denote more refined chunks than those in concern graphs, but concern graph arcs are more specialized.

ConMan is the concern manager part of the Concern Management Environment developed at IBM [9]. Concerns in ConMan can be delocalized, and separate concerns can overlap. Concerns can be associated with program elements either explicitly or implicitly via constraints. The CME project is implemented as an Eclipse plug-in, but the web site indicates that the project was closed in 2006.

2.4 Implementation Approach

Insight into the implementation of holophrastic display for contiguous chunks can be gained from examining how folding is accomplished in Eclipse. Folding in Eclipse's JDT is implemented in a package called JFace [4]. JFace introduces the concept of a *projection document* to intermediate between the actual text file being edited and the graphical user interface (GUI) display widget. Methods are provided to add/remove a subrange of the text occurring in the master document to/from the projection document. That is, chunks are expressed in terms of their contiguous subranges, each of which is announced to the projection document via an explicit method call. Any changes to the master document that affect a subrange are automatically reflected in the projection document and hence in the display.

Coordination between the display and the document is managed by mapping between so-called *widget coordinates* (lines on the screen) and *model coordinates* (lines in the file). Moreover, a range of text as expressed in model coordinates can either be *exposed* or *unexposed*, that is, expanded or collapsed.

Eclipse programmers can configure the units of abstraction and associate visual annotations with them. Callable operations exist for collapsing and expanding individual chunks and expanding all chunks.

3. Research Context

We would like to provide software maintainers holophrastic support for abstracting general regions of source code files. There are two motivating use cases. In the first, a maintainer is confronted with foreign code with the need to understand it in order to perform a maintenance activity. Understanding implies recognizing code chunks and abstracting them away. Abstraction includes selection of the lines comprising a chunk, annotating and then collapsing it. We call such a process *active reading;* that is, the reader does not just passively view the program text, but instead actively documents any recognized abstractions. This use case raises the research question of the extent to which active reading leads to a better understanding of the code than does passive viewing.

The second use case also concerns a maintainer confronted with a maintenance task on foreign code. In this case, the maintainer receives the code in a form produced by the maintainer described in the first use case. That is, the code has been abstracted and documented holophrastically. The use case raises the question of whether performance on the second maintenance task is improved as compared with the situation where the maintainer is provided the code along with traditional documentation.

Underlying the two use cases is the question of the extent to which support for non-contiguous chunking improves program understandability for the maintainers in both use cases as compared with strictly contiguous chunking.

To begin to address these three research questions, we first present a conceptual model that organizes the abstrac-

tions obtained during code reading. The model is described incrementally, beginning with support for contiguous, non-conflicting[2] chunks, then conflicting chunks, and finally non-contiguous chunks. After we have presented the model, we describe the tool, TAJ, we have built that implements it and a pilot study we have conducted examining the first use case described above.

4. Conceptual Model

In this section we present a semi-formal description of an idealized holophrastic tool for viewing contiguous, non-conflicting chunks. We will extend the model in subsequent sections as we add support for conflicts and non-contiguity.

The tool comprises three parts: routines for accessing the source code file to be read by the maintainer, the display widget used to view it, and the internal data representation used to hold the history of folds and annotations. For the time being, we will not be concerned with issues such as whether the tool can be used to edit the file rather than just view it, whether it understands the syntax of the program being viewed, whether abstractions can be spread across multiple source code files, etc. We will also ignore issues such as scrolling and resizing of the display widget.

The input file manager can be modeled as producing a sequence of text lines. Likewise, the display widget needs to present a sequence of lines, some from the input file and some containing user-supplied annotations. Lastly, the internal representation can be modeled as a directed tree. The user interacts with the display by selecting one or more lines and indicating an operation to be performed, either **COLLAPSE**, **EXPAND** or **RECOLLAPSE** (refold an **EXPANDED** fold). In the case of the initial selection of a chunk, the user is asked to provide an annotation. After each operation, the display is updated to reflect the changed state.

We assume but do not model the visual indicator (holophrast) that informs the user of lines on the display that corresponded to collapsed chunks. Also, we assume that obvious errors, such as trying to expand a non-chunk or to collapse without first making a selection, are correctly handled. Finally, we assume for the time being that chunks are contiguous and non-conflicting.

4.1 Directed Tree Model

Given the above assumptions, the state of the internal representation and the meaning of the user operations on it can be represented using a directed tree. The nodes of the tree denote chunks and source code lines, and the branches denote the abstraction relationship between nodes. The tree has a **ROOT** node that represents the entire input file. It has outgoing arcs to child **LEAF** nodes; one corresponding to each line in the file. As the user performs **COLLAPSE** operations, additional, **INTERIOR**, nodes are created. The following properties define the tree model.

- There are three mutually exclusive types of nodes: **ROOT**, **INTERIOR** and **LEAF**. There is exactly one **ROOT** node. The number of **LEAF** nodes is the same as the number of lines in the input file. The number of **INTERIOR** nodes is initially zero and at any given time equals the number of previously executed **COLLAPSE** operations.
- Each node contains three pieces of information: a line of text, a boolean indicating whether the node denotes an **EXPANDED** chunk and a sequence of arcs connecting to child nodes. For a given node, **N**, these pieces are designated respectively as **N.LINE**, **N.EXPANDED** and **N.CHILDREN**.
- For each leaf node **N**, **N.LINE** equals the contents of the corresponding line from the input file. For the **ROOT** node and for each **INTERIOR** node **N**, **N.LINE** corresponds to a user-supplied annotation.
- For each **INTERIOR** or **ROOT** node, the associated **EXPANDED** boolean indicates whether an **EXPAND** (`true`) or a **COLLAPSE** (`false`) operation was more recently performed on that node. **LEAF** nodes always have this boolean `true`.
- The **ROOT** node and each **INTERIOR** node have a sequence of arcs targeted at child nodes. For **LEAF** node **N**, **N.CHILDREN** = <>, the empty sequence of nodes.

Initially, the **ROOT** node has exactly one outgoing arc to each of the **LEAF** nodes; and there are no **INTERIOR** nodes. Moreover, **ROOT.LINE** is the empty line. Also, the **EXPANDED** booleans for all nodes are `true`. The initial situation is depicted in Figure 2. In the drawing, the **ROOT** node is at the top, and **LEAFS** have no descendants. Moreover, each **LEAF** contains the number of the line in the original text file to which the node corresponds. A plus sign (‘+’) within a node indicates that the node is expanded (has its **EXPANDED** boolean `true`). For **INTERIOR** or **ROOT** nodes, a minus sign (‘-’) indicates that the node is collapsed (not depicted in the figure).

4.2 Semantics of Operations

In this subsection we consider the effect on the internal representation of the **COLLAPSE**, **EXPAND** and **RECOLLAPSE** operations. Other operations, such as permanently deleting a chunk, can be similarly defined. The effect of the operations on the display is given in Section 4.3.

2. Two chunks *conflict* if they subsume the same line(s) and are not hierarchically nested. An example is shown in Figure 5 and described in section 5.1.

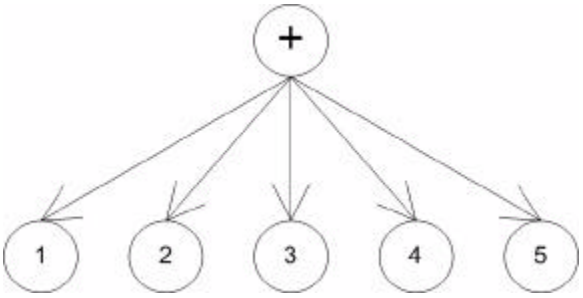

Figure 2. Initial configuration

COLLAPSE. Any non-empty, non-conflicting contiguous sequence of lines from the display that has not been previously folded can be selected. Once the **COLLAPSE** operation is initiated, the following actions take place.

1. An annotation line is solicited from the user.
2. The set of nodes (**NSET**) corresponding to the selected lines is determined.
3. A new **INTERIOR** node (**N**) is created, with its **EXPANDED** boolean `false`.
4. The nearest common ancestor (**NCA**) of **NSET** is determined.
5. Any existing arcs coming into any of the nodes in **NSET** are deleted.
6. An incoming arc to **N** from **NCA** is added.
7. Outgoing arcs from **N** to each of the nodes in **NSET** are created.
8. **N.LINE** is set to be the solicited annotation.

For the situation depicted in Figure 2, if lines 2, 3 and 4 are selected and **COLLAPSED**, the resulting configuration is shown in Figure 3. In the figure, **NCA** is the **ROOT** node, **NSET** is `{2, 3, 4}` and **N** is the node containing the minus sign.

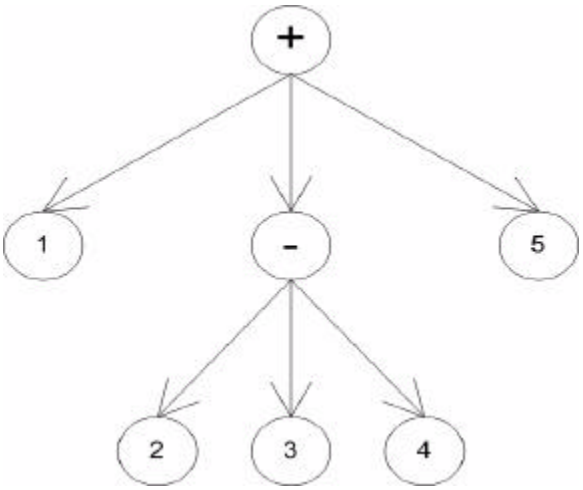

Figure 3. Tree after collapsing nodes 2, 3 and 4

EXPAND. The user can select a line from the display, that represents a previously **COLLAPSED** chunk and request that it be **EXPANDED**. The corresponding node will have its **EXPANDED** boolean set `true`. For the example, if the annotation line denoting the previous **COLLAPSE** is selected, the resulting tree is identical to that in Figure 3 except that the minus sign in **N** becomes a plus sign. Otherwise, the sets of nodes and arcs are unchanged.

RECOLLAPSE. The user can **RECOLLAPSE** a chunk by selecting the corresponding visible lines or their holophrast and requesting a **RECOLLAPSE**. No new annotation is solicited. The only effect on the internal representation is that the corresponding internal node has its **EXPANDED** bit set `true`.

4.3 Display

In addition to the effect on the internal representation, user operations also determine what is visible on the display. Initially, the display is identical to the input. User operations change the state of the internal representation, and the display must be updated to reflect the changed state. At all times the display should accurately reflect the contents of the input and the sequence of user operations. In particular, the following properties should obtain.

1. **Input line integrity.** The order in which any two **LEAF** lines appear in the display is the same as their relative order in the input file. No input line occurs more than once on the display.
2. **Hidden chunk contents.** The line corresponding to a node **N** is only displayed if the path from the **ROOT** to **N** does not go through a **COLLAPSED** node.
3. **Annotation positioning.** An annotation line in the display represents the corresponding lines in the input file. That is, its position in the display relative to other displayed lines should be the same as that of the lines in the chunk it subsumes.

In addition to sustaining these principles, we expect execution of the operations to affect the display as follows.

- **COLLAPSE.** When the user selects lines from the display and requests a **COLLAPSE** operation, the tool solicits an annotation and the selected lines are replaced in the display by the supplied annotation line.
- **EXPAND.** An **EXPAND** operation takes place when the user selects an annotation line that was provided by the user in conjunction with a **COLLAPSE** operation on a set of selected lines. The annotation line on the display is removed, and the corresponding folded lines are redisplayed in its place.
- **RECOLLAPSE**. The **RECOLLAPSE** operator affects the display in the same way that the **COLLAPSE** operator does except that the previous annotation is reused instead of a new one being solicited from the user.

To complete our description of the basic conceptual model, we need to present an algorithm that maps from the internal representation to the display in such a way that the above principles are maintained in the face of user operations.

Mapping. The visual display serves two purposes: it displays the lines from the input file interspersed with annotation lines. Also, it provides a way to map from user-selected lines back into the internal representation. Hence, the display can be modeled as a sequence of nodes. The text lines to be displayed can be extracted from the nodes, and user selections anticipating operations can be mapped to the affected nodes.

Algorithm. This subsection sketches a simple algorithm for updating the display from the intermediate representation. That is, when the user performs an operation and the intermediate representation is updated, the display algorithm is invoked to update the display. The input to the algorithm is the tree; the output is a sequence of nodes to display. The algorithm consists of a simple tree walk. The tree walk is sketched in Figure 4.

```
void walk(Node n) {
  if (n is leaf | ! n.expanded)
    display.add(n);
  else
    for (Node c : n.children)
      walk(c);
}
```

Figure 4. Tree walk to construct display

In the figure, `display` is a global variable whose type is a sequence of `Nodes`. The walk is initiated with a call to `walk(root)`. The algorithm walks the tree but never descends into a subtree whose root is **COLLAPSED**.

5. Extensions to the Basic Model

The model described in the previous section is capable of dealing with non-conflicting, contiguous chunks. In this section, we concentrate on extending the basic model to deal with non-contiguous chunks, beginning with how to deal with conflicts. As it turns out, solving the conflict problem also solves the problem with modeling non-contiguous chunks. Other interesting extensions are discussed in Section 7.

5.1 Conflicting Chunks

Even with contiguous chunks, the above model is insufficient. In particular, it cannot deal with conflicting chunks. This situation does not arise with JDT or with the `#region` directive in C# where all chunks that subsume a given line from the input file must be strictly nested. But sometimes strict nesting is insufficient to document what is going on in the code. Figure 5 depicts such a situation. Note that equally valid situations might have the two **INTERIOR** nodes contain other combinations of pluses and minuses. In the figure, lines 2, 3 and 4 comprise one

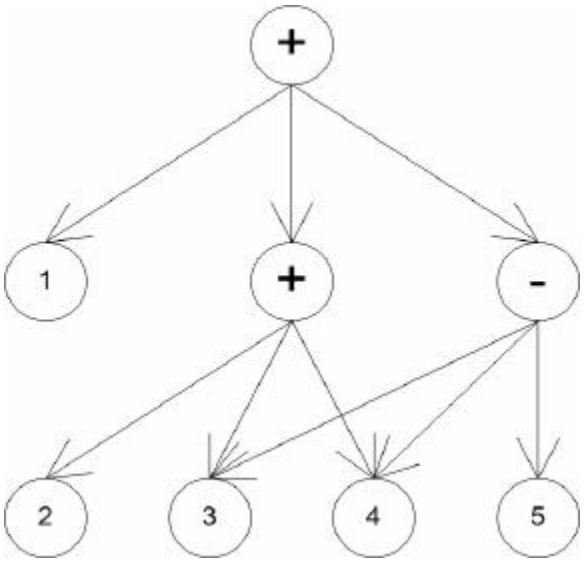

Figure 5. Conflicting chunks

chunk, while lines 3, 4 and 5 comprise another. The former chunk is currently **EXPANDED**, and the latter is **COLLAPSED**.

Both the internal representation and the display algorithm need to be extended to deal with conflicts. For the internal representation, a directed acyclic graph (DAG) is used instead of the tree. This means that a given node may appear in the **CHILDREN** list of more than one **INTERIOR** node.

One other change is made to the internal representation to support the process of walking the graph. Because there now may be more than one path from the **ROOT** to any given node, multiple visits to the node must be prevented. In particular, a new boolean, **VISITED**, is added to each node. Before each walk, all of these booleans are set `false`. Then, each time a node is first reached, the **VISITED** boolean is set `true`. Descent to **CHILDREN** is conditioned on the **VISITED** boolean being `false`, so they are explored exactly once.

The display component is also compromised by conflicts. For example, property 1 (**Input line integrity**) is violated when there is more than one path from the **ROOT** to a given node. Fortunately, the **VISITED** bit readily overcomes this problem.

The second display property (**Hidden chunk contents**) is even more severely affected, and it is not at all clear how to repair it. For example, should node 3 in Figure 5 be displayed or not? One reasonable answer is “no” because the user has chosen to hide it, as indicated by the

COLLAPSE on the rightmost **INTERIOR** node. But the opposite answer is also reasonable because of the **EXPANDED** interior **NODE**. Hence, in adapting property two, different policies are possible.

- Include a node in the results list of the graph walk if any path from the **ROOT** can reach it without going through a **COLLAPSED INTERIOR** node.
- Include a node in the results list only if none of the paths to it from the **ROOT** go through a **COLLAPSED** node.
- Remember the order of **COLLAPSE** and **EXPAND** operations performed by the user and obey the most recently executed one that subsumes the line.
- If we allow different types of annotations, then we may base the decision on the type of a line's annotation, possibly configurable by the user.
- The technology used to display the lines together with user preferences may allow lines to be displayed with different styles, enabling the ambiguous status of a line to be apparent.

Note that the approach described in the previous section effects the first policy above. Other policies require more complex extensions to the graph walk.

Problems with the third property (**Annotation positioning**) are illustrated in Figure 6. The two conflicting chunks share a common initial portion. If both chunks are **COLLAPSED**, which annotation should appear first on the display? If instead one is **COLLAPSED** and the other is not, are both the **EXPANDED** lines and the annotation for the **COLLAPSED** chunk displayed, and in what order?

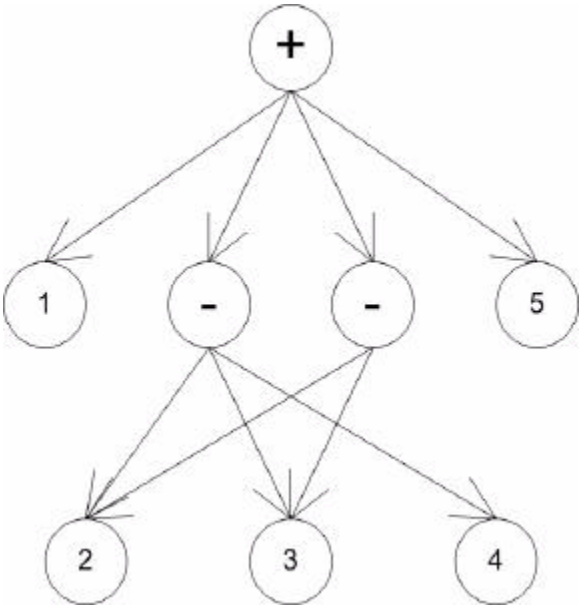

Figure 6. Annotation ordering conflicts

The policy chosen to answer these questions can be effected at the time the chunk is created. Recall that step 6 in the description of the **COLLAPSE** operation given above adds an arc from the nearest common ancestor, **NCA**[3] to the new node. This is equivalent to adding the new node into **NCA.CHILDREN**. Note that the position of the new node within this sequence will dictate the order in which the node is visited during the graph walk. We can express whatever policy we choose for annotation display ordering in terms of that sequence. In the case of Figure 6, if we choose to use the sequence illustrated in the figure, then the annotation will appear in the output display in the order given.

In summary, we can add support for conflicting nodes by adding the **VISITED** boolean, by controlling the order in which nodes are inserted into **CHILDREN** sequences and by enhancing the graph walk to enforce policies in ambiguous situations.

5.2 Delocalized Chunks

A simple example of delocalization is depicted in Figure 7. In the figure, the leftmost chunk, subsuming lines two and four, is delocalized, abstracting **LEAF** nodes 2 and 4. Allowing the user to **COLLAPSE** delocalized chunks

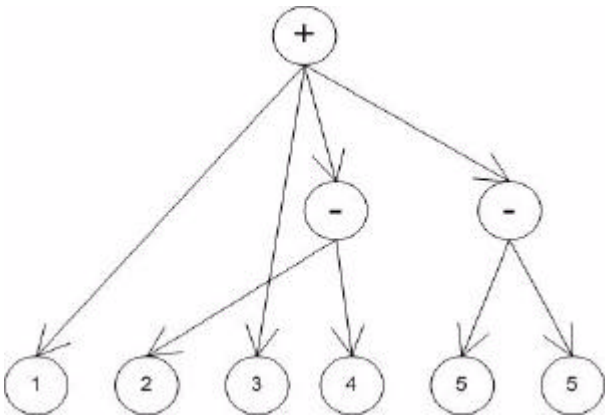

Figure 7. Delocalized chunk

leads to significantly more complex DAGs, but no new representation difficulties. That is, the graph structure and display-update algorithm correctly implement the desired display properties.

6. TAJ

A tool called TAJ has been built to support active reading. Two current versions exist. The simpler of the two supports active reading of a single text file and delocalized chunking. Moreover, annotations and chunking

3. For DAGs the analogous concept is called the *dominator*.

state are persisted so reading sessions can be interrupted. The more complex version currently supports only contiguous chunks, but adds in a linking capability described in Section 7. The remainder of this section describes the implementation of the first version.

The design of the TAJ intermediate representation comprises three classes—`Display`, `Graph` and `Node`—that collaborate via the `Composite, Visitor` and `Observer` patterns from [7]. In particular, the `Graph` class implements the **COLLAPSE**, **EXPAND** and **RECOLLAPSE** operations. Changes to the graph are observed by the `Display` which initiates a graph walk with an attached visitor. The graph is comprised of instances of class `Node` and its subclasses, `LeafNode` and `CompositeNode`.

The current version of TAJ implements the following display policy. If the graph walk reaches a previously unvisited **LEAF** node or a **COLLAPSED INTERNAL** node, then the corresponding contents line is displayed. The implementations of other policies differ in complexity, possibly involving recursive walks of the graph.

Pilot Study. The contiguous version of TAJ described above was used in a pilot study testing the first use case described in Section 3—that active abstraction supported understanding. In the study, twenty-three software engineering graduate students, about half with industrial experience, were provided with an earlier version of TAJ and asked to read and document their understanding of a short (42 lines) but complex and obfuscated Java program. Subjects were instructed to try and understand the program using one of several active-reading strategies. Strategies included manually retyping the program, indenting it, renaming variables with more meaningful names, and providing annotations for chunks. The control group was given no instructions with regard to reading strategy. Understanding was measured at the end of the study period in terms of correct responses to a series of questions about the code. The group that used folding performed better (successfully answered more questions) than any of the other groups, and, in general, the more active the strategy employed, the more correct answers were provided[4].

7. Discussion

There are many interesting extensions to the basic model of holophrastic display described in the previous sections. Section 7.1 describes the design space for holophrastically based software understanding and documentation tools. Section 7.2 briefly describes another implementation of TAJ we have made that supports domain-based program understanding.

7.1 Feature Space

There are many possible extension available in TAJ's design space. The following are simple to implement and ubiquitous.

- Scrolling
- Window resize
- Text editing (may require corresponding modifications to the nodes and arc of the graph)

Also simple to implement, but specific to holophrastic displays are the following.

- Chunk deletion (leaving the underlying **LEAF** nodes)
- Annotation editing

There are also a variety of more complex features necessary when applying TAJ to real-world projects.

- Support for multiple documents, particularly when an abstraction can cross document boundaries
- Multiple annotations for the same chunk, possibly provided by different developers at different times, with accompanying provenance
- Support for programming language syntax including automatic detection of syntactic chunks. Eclipse has mature support for syntactic analysis of Java programs. This can be leveraged to enable more sophisticated abstractions than the line-oriented ones currently supported
- Display order policies, as previously described

It is also interesting to contemplate the following more extrapolative extensions.

- Annotation types, for example, formal pre- and post-conditions
- Interpretation of annotations by external tools in the spirit of Java 5 annotations
- Automatic detection of abstractions and construction of annotations, as, for example, cliche recognition [25] and ownership domains [1]
- Support for a typed abstraction, for example, a design-pattern type
- Interoperation with refactoring tools, providing, for example, the automatic transformation of a chunk using the `ExtractMethod` refactoring [6]
- Metrics and corresponding visualizations, for example, indicating those sections of the code for which relatively fewer annotations have been made
- Validation of abstractions. For example, if the software maintainer has detected a chunk that he or she believes serves the role of a discrimination variable in

4. The preliminary nature of the pilot study and the well-known logistics problems of controlling for independent variables prevent any generalizations being made from the study.

a `switch` statement, make sure that there are no invalidating uses of the variable in the remainder of the code

- Visualizations. Linear text, even when holophrastically managed, is likely insufficient for rendering complex software with multiple, interleaved abstractions. Clearly, the software visualization community has much to offer in supporting this effort.
- Support abstractions over non-text (graphical) documents, such as UML models

7.2 TAJ with Domain Modeling

The second implementation of TAJ is a domain-based program understanding and documentation tool. It is embedded as a plug-in into the Eclipse JDT and supports contiguous, non-conflicting holophrastic chunking of program files. Moreover, it includes a second window containing a graphical model of the program's application domain expressed as a UML class diagram. An existing Eclipse plug-in (EMF[5]) supports the definition of graphical models, and we have used it to enable software engineers to express domain models using a subset of the UML class modeling notation, specifically, classes, methods, attributes and relationships.

Once a user has detected a chunk in the program and provided an annotation, he or she can link that chunk to an element of the domain model. Whereas holophrastic chunking of the program text supports the answering of *what* questions [14], links to the domain model supports answering the all-important *why* questions. That is, a link from a program chunk to a domain abstraction explicitly documents design intent. In particular, this version of TAJ comprises the following features.

- The ability to view a program and abstract it into a hierarchical program model
- The ability to view and extend one or more domain models
- The ability to link program elements to domain elements

In addition to these existing features, we anticipate investigating the following extensions.

- Support for dowsing [3]. *Dowsing* is the process of automatically constructing domain models by analyzing textual descriptions of the domain. We intend to add a dowsing tool into TAJ.
- Re-engineering from procedural programming languages to object-oriented languages. Technically, a domain model in TAJ is an object oriented framework; that is, it is a collection of abstract classes and inter-class collaborations. As such, there is a natural opportunity to use subclass refinement in support of re-engineering from a source program written in a procedural language into an object-oriented version. For example, if a code chunk is detected that corresponds to a domain concept denoted in the domain model with an abstract method, then we can construct a concrete subclass in which the method body corresponds to the (syntactically transformed) chunk contents. Clearly, this step is just part of the overall re-engineering effort, but it can serve as a useful target to regulate the re-engineering process.

8. Conclusion

The grand vision of TAJ is as a software maintenance and documentation environment. An essential part of the tool is the ability to support the maintainer in understanding the source code. Abstraction and, specifically, delocalized abstraction is an essential part of this vision.

Acknowledgements

We appreciate the contributions of Sergio Berzosa Gonzalez, Zaheer Hooda, Jai Kejriwal, Raphael Kobi, Samir Vira and Shaoyu Xue to the early development of TAJ.

References

[1] Marwan Abi-Antoun and Jonathan Aldrich. "Ownership Domains in the Real World". *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2007.

[2] L. A. Belady and M. M. Lehman. Program Evolution. *Processes of Software Change.* Academic Press, 1985.

[3] Richard Clayton, Spencer Rugaber and Linda Wills. "Dowsing: A Tools Framework for Domain-Oriented Browsing Software Artifacts". *Automated Software Engineering*, May, 1998.

[4] Prashant Deva. "Folding in Eclipse Text Editors". `http://www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html`, March 11, 2005.

[5] Alastair Dunsmore, Marc Roper and Murray Wood. "Object-Oriented Inspection in the Face of Delocalization." *International Conference on Software Engineering*, 2000, Limerick, Ireland, pp. 467-476.

[6] Martin Fowler. *Refactoring*. Addison Wesley, 1999.

[7] Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] W. J. Hansen. "User Engineering Principles for Interactive Systems."*Proceedings of the Fall Joint Computer Conference 39"*, 1971, pp. 523-532.

5.`www.eclipse.org/modeling/emf/`

[9] William Harrison, Harold Ossher, Stanley Sutton, Jr. and Peri Tarr. "Concern Modeling in the Concern Modeling Environment." *Workshop for Modeling and Analysis of Concerns in Software (MACS 2005)*, St. Louis, Missouri, May 16, 2003.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin. "Aspect-Oriented Programming". *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997, Jyväskylä, Finland, *Lecture Notes in Computer Science* #1241, Springer-Verlag.

[11] Marcus Knasmüeller. "Reverse Literate Programming." Technical Report CS-SSW-P96-05, Johannes Kepler Universitat Linz, 1996.

[12] Donald E. Knuth. "Literate Programming." *Computer Journal*, 27(2):97-111, May 1984.

[13] Bernt Kullbach and Volker Riediger. "Folding: An Approach to Enable Program Understanding of Preprocessed Languages." *Working Conference on Reverse Engineering*, pp. 3-12, 2001.

[14] Stanley Letovsky. "Cognitive Processes in Program Comprehension". *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, editors, Ablex Publishing, Norwood, New Jersey, 1986, pp. 325-339.

[15] Hanspeter Mössenböck and Kai Koskimies. "Active Text for Structuring and Understanding Source Code." *Software - Practice and Experience*, 26(7):833-850, July, 1996.

[16] Martin P. Robillard and Gail C. Murphy. "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies." *International Conference on Software Engineering, ICSE'02*, May 19-25, 2002, Orlando, Florida, pp. 406-416.

[17] Martin P. Robillard and Gail C. Murphy. "Representing Concerns in Source Code." *ACM Transactions on Software Engineering and Methodology*, 16(1), February, 2007.

[18] Spencer Rugaber. "The Use of Domain Knowledge in Program Understanding." *Annals of Software Engineering*, Volume 9, pp. 143-192, 2000.

[19] Spencer Rugaber, Kurt Stirewalt and Linda Wills. "Understanding Interleaved Code". *Automated Software Engineering*, 3(1/2):47-76, June 1996.

[20] Herbert A. Simon. "How Big Is a Chunk?". *Science*, 183(4124):482 - 488, February 8, 1974.

[21] Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman and Robin Lampert. "Designing Documentation to Compensate for Delocalized Plans". *Communications of the ACM*, 31(11): 1259-1267, November, 1988.

[22] W. P. Stevens, G. J. Myers, L. L. Constantine. "Structured Design." *IBM Systems Journal*, 13(2):115-139, 1974.

[23] David H. Taenzer. "Object-Oriented Software Reuse: The Yoyo Problem". *Journal of Object Oriented Programming*, September-October 1989.

[24] Thomas Teitelbaum. "The Cornell Program Synthesizer". Communications of the ACM, 24(9):563-573, September 1981.

[25] Linda M. Wills. *Automated Program Recognition by Graph Parsing.* Technical report, MIT-AI-TR 1358, MIT Artificial Intelligence Laboratory, Doctoral Dissertation, July, 1992.

[26] Niklaus Wirth, "Program Development by Stepwise Refinement". *Communications of the ACM*, 14(4):221–227, April 1971.