

Fluid Source Code Views for Just In-Time Comprehension

Michael Desmond
University of Limerick &
University of Victoria
michael.desmond@gmail.com

Margaret-Anne Storey
University of Victoria
Victoria, BC Canada
mstorey@uvic.ca

Chris Exton
University of Limerick
Limerick, Ireland
chris.exton@ul.ie

ABSTRACT

The use of modern programming paradigms and technologies, such as object orientation, inheritance, polymorphism and aspect orientation, facilitate a number of important software engineering benefits. Concerns can be better separated, software is more modular and reusable, and evolution is a simpler and less invasive process. However these advances in programming technology also negatively impact the ability of software engineers to read, navigate and comprehend source code. The definition and execution flow of program operations is more fragmented and relationships between software units are often implicit and difficult to determine.

In this paper we propose and present some initial work on a new direction in source code document presentation called *fluid source code views*. Fluid source code views enable programmers working on a primary source code document to fluidly shift attention to related supporting material in a contextual manner. This reduces the need for programmers to navigate and presents code in a manner more efficient and fitting in terms of comprehension. We very briefly discuss the motivation behind fluid source code views, present the approach and discuss both the potential advantages and disadvantages of this technology.

1. INTRODUCTION

Focused code comprehension, the examination of control flow and data manipulation between program units at the source code level, is an important part of many programming tasks. Fixing bugs, adding features or making changes to a software system requires the examination and comprehension of the related source code at a detailed level. However, advanced software abstraction and modularization techniques commonly used today can hinder the task of code comprehension by requiring excessive amounts of navigation, searching and information synthesization during targeted code comprehension.

For example, consider the examination of a medium grain program operation or feature in an object oriented system. The code implementing the feature may well be scattered over dozens of separate files, classes and methods. Parts of the operation may be implemented at various levels in the system type hierarchies and dependent on implicit polymorphic relationships between call sites and implementations. In an AOP [1] environment the implementation may also be dispersed between numerous aspects and advice constructs.

It is fundamental to many software quality measures, such as modularity, reusability, evolvability, non invasive change and separate compilation that program definition is linguistically and physically fragmented into cohesive units with complex relationships. However the negative effect of this increasing fragmentation and complexity is that the programmer must

perform large amounts of code navigation and searching in order to determine how the feature is implemented and how it interacts with other features and software units. Furthermore as a programmer must increasingly navigate through physically separate points in code, loss of context and information synthesization becomes more of a mental burden and distraction making comprehension a confusing and time consuming task.

Modern Integrated development environments (IDEs), such as Eclipse [2], provide tools and views to help programmers find, examine, edit and navigate between points of interest in code. IDEs represent source code primarily in terms of static text documents. Navigation occurs by "jumping" from position in document to document and forces the programmer to mentally maintain context between related points in code.

To ease the constant navigation and mental burden on the programmer during comprehension tasks we present fluid [3] views of source code documents. A fluid source code view is an enhancement of the traditional IDE source code editor with support for fluidly exploring the software space related to the source file being edited. This exploration happens within the context of source file under examination and allows the programmer to fluidly shift attention to semantically related software elements without distraction or loss of context using advanced code "folding" techniques.

In this paper we will describe fluid source code technology and how it is applied to aid in a number of program comprehension scenarios. We describe how fluid source code views can help programmers to comprehend and summarize complex method call chains from the context of a root invocation point. We also look at how fluid views can aid in the comprehension of inheritance hierarchies and polymorphic dispatch sites and how the comprehension of aspect advised classes can be aided by allowing programmers to view the effect of advice in the context of the static join point that is being advised.

Finally we will speculate on the advantages of fluid source code views in terms of code comprehension and also consider some potential issues and limitations with the technology.

2. FLUID SOURCE CODE VIEWS

A fluid source code view is identical to a regular source code editor but contains a sophisticated interactive software space explorer seamlessly built into the code itself.

When a programmer is examining a source code document using a fluid source code view they can interact with certain semantically important points within the code to reveal related information that supports common comprehension tasks and patterns.

2.1 FLUID DOCUMENT TECHNOLOGY

Fluid document technology recognizes that documents typically consist of primary information and supplemental linked or related material. For instance a source code document contains primary code contained within the document itself and may also contain explicit and implicit links to related code such as method invocations, advising aspects and inherited super class behavior.

It is common that programmers examining primary material would like to examine related material, but to do so without leaving the context of the primary document. Fluid documents provide unobtrusive, optional, contextual (sometime animated) access to supporting information, allowing the reader to fluidly shift attention from primary to supplemental information and back again to the primary information. Fluid documents support greater engagement with primary material by enabling non disruptive just-in-time access to related content. Fluid source code views are the application of fluid document technology to source code documents. The fragmented and linked nature of software means that it is an ideal medium to be supported by fluid document technology.

2.2 EXAMPLES

To explain fluid source code views in a succinct manner, we use the following examples.

2.2.1 METHOD CHAINS

The first application of fluid source code views is the comprehension of control and data flow between method chains. We use the term method chain to describe a root method calling a number of sub methods, which in turn calls a number of sub methods and so on, in essence a rooted call hierarchy.

In a traditional source code editor a programmer undertaking the simple task of examining the control and data flow between a number of methods must examine each method in isolation, regularly jumping back to previous methods to follow alternative control paths or to clarify information from the calling context. While this seems natural, as it reflects how the software is modularly structured, it would be more fitting to the conceptual model maintained by the programmer and their flow of concentration if all the methods were available for viewing in the same window, so that the overall control and data flow could be examined (see Fig. 3).

When a source code document is opened in a fluid source code view, each method invocation contained in the document is annotated with an unobtrusive visual cue (see Fig. 1).

```

completeInitialization();
initializeFluidDocument(getDocument());

```

Figure 1: Unobtrusive visual marking of method invocation sites

When the programmer moves the mouse to within the bounding region specified by the visual cue an expandable widget appears and the expandable method invocation is underlined with a visual cue (see Fig. 2). The underlining of the expandable method invocation is useful to distinguish the “owner” of an expandable widget in the case of method invocation nesting, where expandable widgets can be grouped quite close to each other.

```

completeInitialization();
initializeFluidDocument(getDocument());

```

Figure 2: Revelation of the expandable widget and visual indication of the expansion target

Each widget occupies a single character in the source code document and is completely unobtrusive to the normal functioning and editing features supported by the code editor. Using the caret the programmer can edit around and directly upon the visual cue, the fluid document editor takes care of repositioning the visual cues in response to document edits and the addition and removal of cues that are no longer valid expansion points.

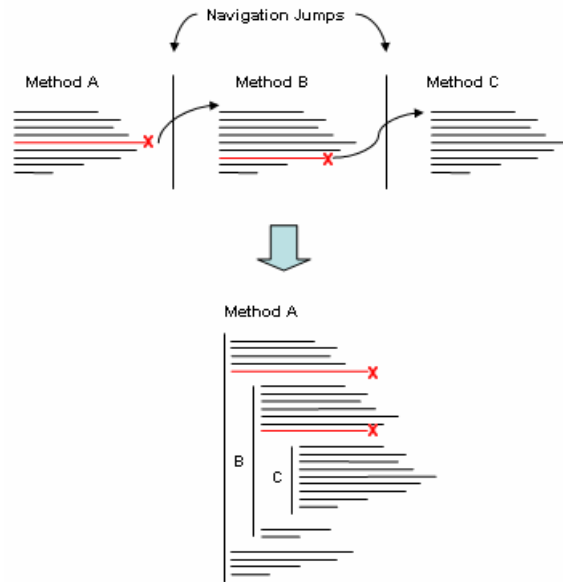


Figure 3: breaking method chain modularity for enhanced alignment with programmer cognition

When a method invocation widget is clicked it will expand: expansion of the method invocation widget reveals a simulated textual representation of the target method definition in a semantically accurate context within the current source code document (see Fig. 4).

The related information, in this case the method definition, is colored to indicate that it is supplemental to the primary document. The expanded method definition also contains any leading comments connected with the method. We intend to make these comments collapsible in a future version. We also intend to make the presentation of the method signature and surrounding braces an optional feature of the expanded code. To navigate to the document within which the method definition is contained the programmer may hold the control key and click within the exposed region.

```

o  completeInitialization()
  /**
   * Initializes document listeners, positions, and position updaters.
   * Must be called inside the constructor after the implementation plug-ins
   * have been set.
   */
  protected void completeInitialization() {
    #Positions= new HashMap();
    #PositionUpdaters= new ArrayList();
    #DocumentListeners= new ArrayList();
    #RenotifiedDocumentListeners= new ArrayList();
    #DocumentPartitioningListeners= new ArrayList();
    #DocumentRewriteSessionListeners= new ArrayList();

    addPositionCategory(DEFAULT_CATEGORY);
    addPositionUpdater(new DefaultPositionUpdater(DEFAULT_CATEGORY));
  }
  initializeFluidDocument(getDocument());

```

Figure 4: Expansion of a method invocation widget reveals the target method definition in context

Any valid method invocations within the newly expanded method definition are also available to be expanded and thus the programmer can further selectively expand deeper into the call hierarchy, keeping a constant visual representation of the flow of control and data through the various methods. This feature of fluid source code documents reduces the need for navigation between multiple documents and supports just-in-time comprehension of the control flow related to a particular root method definition. A programmer using this feature need not synthesize information from method call to method call as this information is visually displayed in the fluid source code view. When the programmer is finished examining the expanded code it can be collapsed by clicking its widget again. Note that exposed code is not editable however as our prototype becomes more advanced we may experiment with this feature.

2.2.2 INHERITANCE

The next feature of fluid source code views that we examine is support for working with inheritance relationships at the source code level. To comprehend the definition of a subclass method that overrides a super class method it is sometimes necessary to comprehend the super class implementation and so on up the hierarchy. This occurs due to the close coupling between super and subclass. However, due to the modular structure of code, the programmer is forced to navigate up the class hierarchy examining each level of implementation in isolation.

When a sub class is opened in a fluid source code view any super method calls are treated as expandable and thus the programmer can reveal the effect of the various levels of super class implementations on the current subclass context. Again this reduces the need for bothersome navigation up through the inheritance hierarchy and allows the programmer to view from the context of the subclass method the effect of super class coupling on control and data flow (see Fig. 5). Again any further expandable points within an exposed super class range are available for nested expansion.

Sometimes super class methods are overloaded without the calling of the super class method itself within the subclass implementation. There is usually a good reason for this particular situation. When viewing source code using a fluid source code view, the programmer can choose to view the super class implementation of a method even when it is not explicitly called. This enables the programmer to examine and compare the overridden method in context with its replacement and thus shed light on the programming decision which resulted in the organization of code in this way. The widget indicating that this

feature is available for a particular method definition is a green upwards facing arrow located at the start of the method signature.

```

o  public FluidAnnotation(boolean isCollapsed) {
  super(TYPE, false, null);
  /**
   * Creates a new annotation with the given properties.
   *
   * @param type the type of this annotation
   * @param isPersistent <code>>true</code> if this annotation is
   *         persistent, <code>false</code> otherwise
   * @param text the text associated with this annotation
   * @since 3.0
   */
  public Annotation(String type, boolean isPersistent, String text) {
    #Type= type;
    #IsPersistent= isPersistent;
    #Text= text;
  }
  #IsCollapsed= isCollapsed;
}

```

Figure 5: Working with inheritance relationships using fluid source code views

2.2.3 POLYMORPHISM

Fluid source code views analyze the static type structure of the software being examined and provide a set of potential matches at polymorphic dispatch sites. When a polymorphic dispatch site is encountered, it is annotated with a special widget similar to the widget used to annotate monomorphic dispatch sites. However when a polymorphic dispatch site is "expanded" a selection of concrete implementations is revealed and the programmer can then decide which concrete implementation they would like to view in context (see Fig. 6).

```

o  IShape shape = getSelectedShape();
o  Rectangle bounds = shape.getBounds();
  Circle.getBounds();
  Triangle.getBounds();
  Ellipse.getBounds();

```

Figure 6: Fluid view of polymorphic dispatch site

The presentation of polymorphic call sites in this manner means that programmers need not leave the comprehension task at hand to examine the type hierarchy but can instead view the information they require without interruption. This view also allows the visual comparison of concrete implementations within a single editor window.

2.2.4 ASPECT ORIENTED PROGRAMMING

Fluid source code views support comprehension of the implicit relationship between classes and advising aspects. When an advised class is opened using a fluid source code view, it is analyzed for crosscutting structure, and the static join points contained within the class that are targeted by advice are marked with an expandable indicator. When the indicator is expanded, it signals that the programmer wants to examine the advice potentially affecting the join point, and the viewer displays an ordered selection of advice elements. The programmer can then select particular advice elements for expansion thus revealing the extra control flow that is executed at the join point (see Fig. 7).

```

/**
 * @param x
 * @param y
 */
public void performMove(int x, int y)
{
    before(Piece piece, int x, int y):movement(piece,x,y)
    {
        System.out.println("Move " + piece + " to " + x + " " + y );
    }
    Move move = new Move(this,x,y);
    fireMoveEvent(move);
}

```

Figure 7: Fluid representation of advice

Because the application of advice to a static join point can be guarded by a dynamic check, we are working on presenting advice at join points in a way that indicates firstly that dynamic checks may affect the application of advice to a join point and secondly what dynamic checks are considered when determining the application of advice. Prior Experiments applying fluid document technology to AOP are discussed here [4].

In the current prototype advice is represented by an icon within the gutter region of the target join point, in a future version we plan to devise in-code cues and widgets to allow programmers better work with and visualize advice at join points.

3. FLUID SOURCE CODE VIEWS AND CODE COMPERHENSION

We speculate that fluid source code views will help in the task of software comprehension by reducing navigation, loss of context and the need to synthesize information from disjoint points in code. We believe that fluid source code views provide cognitive support for tasks such as following control and data flow through software and understanding, in a practical way, how fragmented pieces of software are related and affect each other.

It is worth noting that although fluid source code may help programmers navigate and comprehend code it may also foster some negative assumptions about software abstraction. It has been noted [4] that fluidly viewing aspect advice can encourage programmers to think of AOP in terms of ‘injecting’ code at join points. This is contrary to the correct understanding of aspects as a first class structure of a software system and their linkage to dynamic points of execution. We also recognize that fluid source views may also foster the comprehension of inheritance in terms of code structure as opposed to a conceptual model of software organization.

4. FLUID ISSUES

The application of fluid document technology to source code is not without its issues. As we have thought about and experimented with the technology we have considered many potential problems. As mentioned in the previous section, fluid source code views may encourage programmers to think about software paradigms and concepts inaccurately. However we believe that a solid understanding of abstraction and an acceptance that fluid views are simply a convenient visualization of software documents will alleviate this potential issue.

The presentation aspects of fluid source code is also an active part of our research, it can be difficult for a programmer to differentiate and recognize the boundaries between primary content and related content when large amounts of related content is visible, this is especially true with code as it all looks alike. Views can easily become over cluttered. Also the expressive

nature of software definition means that links to related information can sometime occur in awkward positions. We are investigating the use of various presentation and interface technologies such as color, opacity and animation in order to better enable programmers to deal with the use of fluid document technology with source code.

5. CONCLUSION

We believe that source code is an excellent medium for experimentation with fluid document technology. We are currently working on the prototype of fluid source code views using the Eclipse development environment as our medium, the prototype is a work in progress. Our future work will involve the completion and release of a working prototype, an evaluation of the impact of fluid source code views on software comprehension and the investigation of further applications of fluid document technology within the software domain.

6. REFERENCES

- [1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V.Lopes, J-M Loingtier, and J. Irwin. Aspect-Oriented Programming. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), pages 220 – 242, Jyväskylä, Finland, 1997.
- [2] The Eclipse project, <http://eclipse.org>
- [3] Chang, B.W., Mackinlay, J.D., Zellweger, P.T. and
- [4] Mik Kersten, Matt Chapman, Andy Clement, Adrian Colyer: Lessons learned building tool support for AspectJ. AOSD 06.