Research in Coding and IDEs

Jan-Peter Krämer
Media Computing Group
RWTH Aachen University

http://hci.rwth-aachen.de/cthci
Time in Software Development

[LaToza2006, Maintaining mental models: a study of developer work habits]
Software is complex and hard to understand.
Task context

- What is relevant information?
- What strategies are applied to find information?
Models for Developer Strategies

[Ko2006, An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks]

31 Professional Java Developers

5 Maintenance tasks
(3 Bugs, 2 Enhancements)

500 SLOC Java Paint Application
Models for Developer Strategies

[Ko2006, An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks]
Models for Developer Strategies

[Sillito2008, Asking and Answering Questions during a Programming Change Task]

9 experienced developers (pair programming)

1 of 5 maintenance tasks per session

ArgoUML 60k SLOC

16 developers from industry

Real world change task

Real world sour code
Models for Developer Strategies

[Sillito2008, Asking and Answering Questions during a Programming Change Task]

- Finding focus points
- Expanding focus points
- Understanding a subgraph
- Questions over groups of subgraphs
package org.jhotdraw.contrib;

import org.jhotdraw.framework.DrawingView;

/**
 * Some events require the previous DrawingView (e.g. when a new DrawingView
 * is selected).
 */
private DrawingView myPreviousDrawingView;

public class DesktopEvent extends EventObject {
    private DrawingView myDrawingView;

    public DesktopEvent(Desktop newSource, DrawingView newDrawingView) {
        this(newSource, newDrawingView, null);
    }

    public DesktopEvent(Desktop newSource, DrawingView newDrawingView, DrawingView newPreviousDV) {
        super(newSource);
        setDrawingView(newDrawingView);
        setPreviousDrawingView(newPreviousDV);
    }

    private void setDrawingView(DrawingView newDrawingView) {
        myDrawingView = newDrawingView;
    }

    public DrawingView getDrawingView() {
        return myDrawingView;
    }

    private void setPreviousDrawingView(DrawingView newPreviousDrawingView) {
        myPreviousDrawingView = newPreviousDrawingView;
    }

    public DrawingView getPreviousDrawingView() {
        return myPreviousDrawingView;
    }
}
Tools Used in Eclipse

[Murphy2006, How Are Java Software Developers Using the Eclipse IDE?]

Table 2 lists the commands by the number of developers using the command. Table 3 lists the commands according to average use by all developers. Interestingly, developers used content assist (which suggests possible method names in the editor given a type) as much as the common editing commands.

Analyzing the command information in the interaction histories was difficult. For Eclipse and the plug-ins that extend it, the intent for the plug-in developer is to assign a unique identifier for a command regardless of how the command is made available in the environment. For instance, the same command provided through a toolbar menu and a context menu in the editor should have the same identifier. Unfortunately, not all Eclipse plug-ins use this convention. As a result, we found many inconsistencies, resulting in different identifiers representing the same command. For example, selecting Save from the File menu in the toolbar generates a different identifier than when a key binding performs the Save command. We also found cases that used the same identifier for commands provided by different plug-ins.

To account for these duplications and ambiguities, we created a mapping of identifiers that considers the context of how a command was used. This mapping reduced the number of unique identifiers from 1,208 to 1,142. However, our mapping focused on the more commonly used commands, so this number might still include duplicated and ambiguous commands. To facilitate this sort of analysis, we recommend that plug-in developers specify consistent IDs for their commands and actions.

Navigation

Most software fixes, changes, and enhancements involve navigating across the code base to understand the system's structure and the context in which code executes. Eclipse provides seven views to help a developer efficiently locate code of interest: Package Explorer, Type Hierarchy, Outline, Search, Call Hierarchy, Bookmarks, and Declaration. The developers in our study used the Package Explorer view the most, on the basis of the number of selections made in each view (see figure 5); nobody used the Declaration view, even though it is present by default in the Java perspective.

Through key bindings, Eclipse also provides direct, easily accessed support for different kinds of nonlocal navigation and searches, including navigating to the declaration of an element selected in the editor, searching for references to a selected element, and opening a type. Table 4 summarizes these nonlocal navigation and search commands available in the JDT, their key bindings on the Windows platform, how many of the developers used the commands, and each command's rank (a rank of one indicates the command that the developers used most; the lowest rank is 1,142—the number of commands). This data shows that the command used most often is opening a selected element's declaration (a rank of 21); the command used by the largest number of users is the search for references in a workspace.

To help developers mark points of interest
Easing Access to Task Context
[Kersten2006, Using Task Context to Improve Programmer Productivity]
Recommender Tools

[Singer2005, NavTracks: supporting navigation in software maintenance]
[DeLine2005, Easing program comprehension by sharing navigation data]
[Čubranic’2005, Hipikat: recommending pertinent software development artifacts]

• Calculate a Degree of Interest for source code elements based on:
  • reading history
  • editing history
  • history of other team members
  • information from version control systems

• Remaining Problems:
  • Still only text-based visualization
  • Recommendations for irrelevant code are still irrelevant

![Image of Visual Studio interface with Class View and Package Explorer]

As the event stream is constructed, redundant events are filtered. Currently two types of events are removed: metadata and navigational errors. Analysis of the remaining event stream can be refined. Moreover, new filters can be added. If the new filters turn out to be inaccurate they can easily be removed. More study of the navigational errors and that the one second threshold may be too high or too low.
Changing the Presentation


To navigate using the CT Scrollbar, a developer can either use the scrollbar at left in the usual way, or she can click on a location in the thumbnail to jump to the corresponding place in the code. Whenever the mouse cursor is inside the thumbnail area, labels appear showing the names of likely navigation targets, specifically the names of second-level items with no children (e.g., enums) and third-level items (fields and methods) as shown on the right side of Figure 1. In the current design, these pop-up labels occlude the code shape, which is an area for improvement.

The CT Desktop, shown in Figure 2, shows a thumbnail image of every source file in the project, arranged on a desktop surface. Each thumbnail has a label at the top, which shows the file name and serves as a handle for moving the thumbnail. A developer can arrange the thumbnails on the desktop as she sees fit. The code thumbnails are drawn exactly like those in the CT Scrollbar, except that the currently visible portion is drawn with a filled rectangle to make it more apparent. We use the same font size for all thumbnails on the desktop, which means that each thumbnail's height is proportional to its file's length. The document whose editor is active is highlighted with a thicker border than the others. Documents that are currently closed are shown with a grey background, grey title and no scroll area. As with the CT Scrollbar, moving the cursor over a thumbnail pops up target labels, and clicking on a thumbnail activates the document's editor and scrolls to the chosen part of the document. Clicking a thumbnail's title area activates the document's editor without scrolling the document. Double-clicking a grayed thumbnail opens the document and activates its editor.

When the programmer uses any of the standard search tools, the search results are highlighted in both the CT Scrollbar and CT Desktop. This makes it easy to see all search results at a glance.

Both the CT Scrollbar and Desktop are intended to allow the developer to form spatial memory of the code. The CT Scrollbar provides a stable, one-dimensional space per document, with visual landmarks to help the user distinguish different parts at a glance (namely, the code shape, the brackets and the target labels). The CT Desktop provides a stable, two-dimensional space of all the documents, again with visual landmarks (namely, the thumbnail landmarks, plus their placement).

Our UI design choices were driven by our study goals. Specifically, we were interested in whether developers could form spatial memory of the code and how that would affect their navigation choices.
[Bragdon2010, Code bubbles: a working set-based interface for code understanding and maintenance]
[Bragdon2010, Code bubbles: a working set-based interface for code understanding and maintenance]
[Bragdon2010, Code bubbles: a working set-based interface for code understanding and maintenance]
Bragdon2010, Code bubbles: a working set-based interface for code understanding and maintenance
Canvas Interfaces in the Wild
[DeLine2012, Debugger Canvas: Industrial experience with the code bubbles paradigm]
Utilizing the Call Graph
For task 2, Whyline participants viewed significantly fewer files per minute and overall, by task and condition. For task 1, Whyline participants viewed 1 or 2 “why did” questions, almost exclusively about the creation of the list of objects containing the labels (which was a few dependencies away from the bug). Because the task was more difficult, both groups experienced ceiling effects, causing no difference in speed. There was no relationship between industry experience and success for either task (though the sample size was probably too small to detect such differences).

The tools were instrumented to capture data about source file views and navigations with participants explored. The tools were instrumented to record the number of files visited per minute and overall, the median distance to the key function for each task. For task 1, Whyline participants were significantly closer to the key function than the control group (t=4.6, df=18, p<.0002). For task 2, participants were significantly closer to the key function for each task. For task 1, Whyline participants viewed more files overall. For task 2, Whyline participants viewed significantly more files overall. For task 2, Whyline participants viewed fewer files than the control group (t=2.2, df=18, p<.05). This discrepancy is consistent with the nature of the two tasks: task 1 involved changes to the UIs used to debug. As seen at the bottom of Table 1, there was no significant difference in distance (likely due to our extensive 3-month period of user testing prior to the study).

The enthusiasm of participants was clearly evident and all asked to be notified of the tool's availability. Finally, 8 of the 10 Whyline users offered their opinions on the tool. Their feedback was generally positive, with comments such as “This is really going to reduce the burden on programmers.” “I think this will really help.” “This is so cool.” and “My god, this is so cool.” It's so nice and straight and simple...
In practice: Feasible paths most interesting

[LaToza2010, Developers ask reachability questions]
Utilizing Call Graph Information

[LaToza2010, Searching Across Paths]
Static Analysis in the Wild

[Clang Static Analyzer, http://clang-analyzer.llvm.org/]
Call Hierarchy
Stacksplorer

[Karrer2011, Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency]
Blaze

[Krämer2012, Blaze: Supporting Two-phased Call Graph Navigation in Source Code]
Analyzing Navigation Behavior
Information Foraging Theory

Predator

Scent

Prey
Information Foraging Theory

[Lawrance2010, Reactive information foraging for evolving goals]
<table>
<thead>
<tr>
<th>Find Change Location</th>
<th>Task Success</th>
<th>Task Completion Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xcode</td>
<td>Call Hierarchy</td>
<td>Stacksplorerer</td>
</tr>
</tbody>
</table>

Side Effects of Change

33 Developers

80,000 Lines of Code

[Krämer2013, How Tools in IDEs Shape Developers' Navigation Behavior]
Task Success

\[ p = 0.015 \]

<table>
<thead>
<tr>
<th>Tool</th>
<th># of Successful Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xcode</td>
<td>0</td>
</tr>
<tr>
<td>Call Hierarchy</td>
<td>4.5</td>
</tr>
<tr>
<td>Stacksplorer</td>
<td>6</td>
</tr>
<tr>
<td>Blaze</td>
<td>4.5</td>
</tr>
</tbody>
</table>
Task Completion Time

$p=0.022$

<table>
<thead>
<tr>
<th>Tool</th>
<th>Total Time Required in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xcode</td>
<td>1900</td>
</tr>
<tr>
<td>Call Hierarchy</td>
<td>1425</td>
</tr>
<tr>
<td>Stacksplorerer</td>
<td>950</td>
</tr>
<tr>
<td>Blaze</td>
<td>475</td>
</tr>
</tbody>
</table>
Effectiveness

Xcode

Call Hierarchy

Stacksplorer

Blaze

Efficiency

Xcode

Call Hierarchy

Stacksplorer

Blaze

Why?

UI Differences

Navigation Behavior
[Fouse2011, ChronoViz: A system for supporting navigation of time-coded data]
Comparing Navigation Behavior
\[ I_1 = (p_{1,1}, \ldots, p_{640,480}) \]

\[ I_2 = (p_{1,1}, \ldots, p_{1024,768}) \]

I. Features

2. Transformations
\[ H=(m_1, \ldots, m_i) \]

[Piorkowski2011, Modeling programmer navigation: A head-to-head empirical evaluation of predictive models]
A Predictor

[Piorkowski2011, Modeling programmer navigation: A head-to-head empirical evaluation of predictive models]

\[ H = (m, a, b, a, d) \]

**Navigation History**

**M**

All methods known to developer at time \( i \)

**A**

Activation value for each method in \( A \)

**R**

Rank-transformed version of \( R \)

Result: \( N \) top-ranked methods
Prediction Accuracy

Xcode

Stacksplorer

Recency
Frequency
Fwd Call Depth
Undirected Call Depth
Working Set
Bug Report Similarity
Within File Distance

N = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Prediction Accuracy

Xcode

Stacksplorer

\[ Fwd \text{ Call Depth} \]

\[ Undirected \text{ Call Depth} \]
Away from static analysis only
```

<mx:Script>

  <![CDATA[
    public function loadData():void {
      URLLoader
    }
  ]]>"

</mx:Script>
</mx:Application>
```
// Introducing Codelets...
```javascript
// tree

function drawTree () {
    var blossomPoints = [];
    resetRandom();
    drawBranches(0, -Math.PI/2, canvasWidth/2, canvasHeight, 30);
    resetRandom();
    drawBlossoms(blossomPoints);
}

function drawBranches (i, angle, x, y, width, blossomPoints) {
    ctx.save();
    var length = tween(1, 1.60, 12, 3) + random(0.7, 1.3);
    if (i <= 0) { length = 97; }
    ctx.translate(x, y);
    ctx.rotate(angle);
    ctx.fillStyle = "#0000";
    ctx.fillRect(0, -width/2, length, width);
    ctx.restore();
    var tipX = x + (length - width/2) + Math.cos(angle);
    var tipY = y + (length - width/2) + Math.sin(angle);
    if (i <= 4) {
        blossomPoints.push([x, y, tipX, tipY]);
    }
    if (i <= 6) {
        drawBranches(i + 1, angle + random(-0.15, -0.05) + Math.PI, 
        drawBranches(i + 1, angle + random(0.15, 0.05) + Math.PI, 
    } else if (i <= 12) {
        // continue
    }
}
```
Demo
Live Coding Affects Coding Behavior

[Krämer2014, to appear, How Live Coding Affects Developers’ Coding Behavior]

Average Bug Fixing Time

- Control
- Live Coding

- 1200
- 900
- 600
- 300
- 0
Summary

Finding focus points

Expanding focus points

Understanding a subgraph

Questions over groups of subgraphs