Code Mixer: A Visual Approach to Code Comprehension and Information Foraging

Master's Thesis at the
Media Computing Group
Prof. Dr. Jan Borchers
Computer Science Department
RWTH Aachen University

by
Ardi Tjandra

Thesis advisor:
Prof. Dr. Jan Borchers

Second examiner:
Prof. Dr. Horst Lichter

Registration Date: April 19th, 2013
Submission Date: October 7th, 2013

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, September 2013
Ardi Tjandra

Contents

List of Figures

List of Tables

Abstract

A software maintenance task can typically be divided into two phases: *code comprehension* and *information foraging*. During the former, developers try to grasp the rationale behind source code, while in the latter developers performs code modification, which is often accomplished by integrating others' code examples that are found on the Web. Many problems arises during this period, and accordingly, plenty of tools have been proposed to solve them. However, most of these tools only deal with one of the aforementioned phases, forcing developers to utilize a lot of tools to fix different kinds of problems. To this end, in this thesis work we present the design for Code Mixer, a tool that should help developers throughout the entirety of the maintenance period. At the center of Code Mixer are *compositions*, which is a concept that we utilize to represent both the problems that developers face and the pieces of information that they share among one another. Furthermore, Code Mixer is based on three design principles (visual emphasis, portability and task orientation), further ensuring its effectiveness as a help tool. To formulate the design, we underwent a progressive design period that spans four phases. Code Mixer stared off as a web-based environment that differentiates between two user roles, composers and consumers. However, it ended up being a system that combines a desktop interface and a central repository, with unified user experience for all users, making it possible to grow into a thriving community where users can share ideas in the form of compositions. To evaluate our design we conducted a set of qualitative users studies, where users gave unanimously positive reactions regarding Code Mixer's ability to provide assistance for developers during the maintenance period. Ultimately, what separate Code Mixer from the other help tools is the way it bridges the gap between code comprehension and information foraging. It addresses information foraging by supplying a standardized format for task description in the form of design patterns-like structure, on top of providing lightweight creation, distribution and reuse of compositions. Meanwhile, it helps code comprehension by providing a task-oriented outlook of source code, as well as allowing for an efficient navigation through code that focuses on the current task.

Acknowledgements

Working on this thesis has been a rewarding, often illuminating experience, and it would not be possible without the help of the people around me. On that account, I would like to extend my gratitude to:

Leonhard Lichtschlag, whose patient guidance and insightful advice really helped mold this thesis work into what it is.

Moritz Wittenhagen, for his valuable pointers and keen eyes.

Prof. Dr. Jan Borchers, my thesis advisor, for helping to establish the foundation of this research.

Prof. Dr. Horst Licher, for his time and support as an examiner of this thesis work.

The assortment of sharp and colorful folks at the Media Computing Group, for various assistance, insights and inspirations.

The participants to my user studies, for their time, efforts, and valuable inputs.

And finally, my family, for their continued love and support.

Conventions

Throughout this thesis we use the following conventions.

Text conventions

Definitions of technical terms or short excursus are set off in coloured boxes.

EXCURSUS:
Excursus are detailed discussions of a particular point in a book, usually in an appendix, or digressions in a written text.

Definition:
Excursus

Source code and implementation symbols are written in typewriter-style text.

`myClass`

The whole thesis is written in Canadian English.

Download links are set off in coloured boxes.

File: myFile[a]

[a] http://hci.rwth-aachen.de/public/folder/file_number.file

Chapter 1

Introduction

In order for a software to stay functional for as long as possible, it needs to have the ability to adapt to an assortment of unpredictable conditions such as errors and bugs, as well as the changing needs of its users. Therefore, software maintenance is a crucial period of a software life cycle, and can last as long as the life duration of the software itself.

Software maintenance is an important part of a software life cycle

Maintaining a software typically involves *code modification*, which can range from altering only a few lines of code to multiple code blocks spanning multiple code files. Particularly when dealing with cases such as the latter, developers are required to have a good grasp of the code that they are about to change, as careless modification may introduce bugs that could potentially affect the whole system. As a code project grows in size and intricacy, the process of understanding code would become harder and harder to accomplish. For this purpose, a wide array of tools, often called *Software Visualization Tools*, have been proposed, offering a varied set of features to help developers familiarize themselves with source code.

Code modification often requires developers to first understand the code that they are about to change, which is not always easy to accomplish.

However, understanding source code is only half the battle. The expansive, ever-growing nature of programming languages frequently presents developers with technical details that they are not familiar with. The Internet is a valuable resource for this sort of problems, since it hosts numer-

The Internet can be used to solve technical limitations, but it poses a new set of problems.

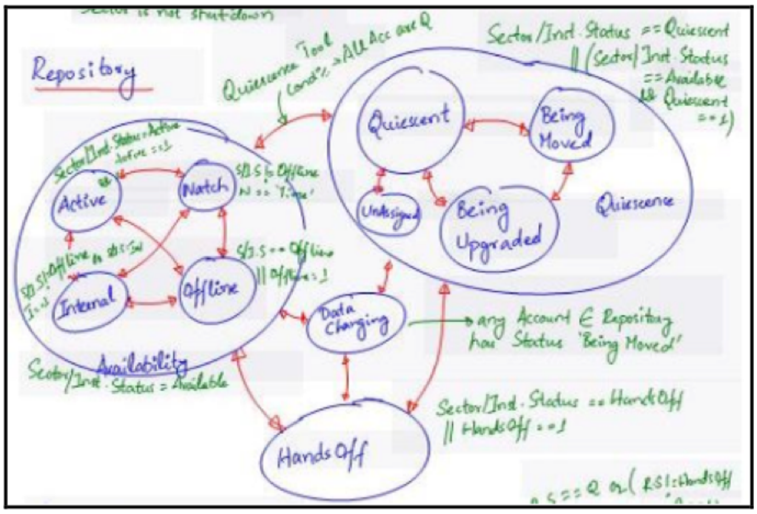

Repository
Quiescence Tool Cond'n => All Acc are Q
Sector/Ind. Status == Quiescent || (Sector/Ind. Status == Available && Quiescent == 1)
Quiescent
Being Moved
Unassigned
Being Upgraded
Quiescence
Active
Watch
Internal
Offline
Availability
Sector/Ind. Status = Available
Data Charging
any Account ∈ Repository has Status 'Being Moved'
Sector/Ind. Status == HandsOff || HandsOff == 1
Hands Off

Figure 1.1: A sample sketch by developers on a whiteboard. Most resulting sketches contain graph-like structures. Figure taken from Cherubini et al. [2007].

ous amounts of tutorials and code examples that developers can use for a lot of purposes, from learning new techniques to borrowing existing solutions to save time. It does have its own set of complications, however, as it is typically difficult to locate the most relevant information within the Internet's seemingly infinite space. Furthermore, reusing code blocks, particulary those created by others, can also raise a lot of new problems that require as much effort to fix as to build said block from scratch. Accordingly, many different tools and algorithms are readily available to help ease up code modification tasks.

Developers are not inclined to make use of the available help tools.

Curiously, a finding by Roehm et al. [2012] suggests that despite the availability of these help tools, developers often default to the more conventional method of using manual *sketches*, particularly when trying to understand code. It seems that developers may be overwhelmed by the sheer number of available tools, each with its own specific techniques and purposes, that they ultimately decide to forgo investigating these tools in the first place. The fact that some of these tools also demand too much effort to get used to does not help matter, as it would discourage developers from actively taking advantage of their various features.

1.1 Objectives

This thesis work aims at formulating a design for a tool that offers a more comprehensive assistance for developers during software maintenance. The tool should ideally be able to cover all phases of a maintenance task, from understanding source code to the actual modification of the code, including looking for, retrieving and integrating relevant information gathered from the Internet. An all-encompassing tool would diminish the need to deal with a big number of tools, and therefore developers would be encouraged to actually use it.

The main objective is to propose a design for a tool that offers a comprehensive help for developers.

To further increase the usability of such tool, the proposed design should exhibit a degree of simplicity that would allow it to accommodate developers with varying skill levels. One possible approach is by taking advantage of the various schemes that software developers are already familiar with. For instance, the proposed design should capitalize on developers' inclination to use sketches, which means that it should allow the users to freely construct images based on their own whims.

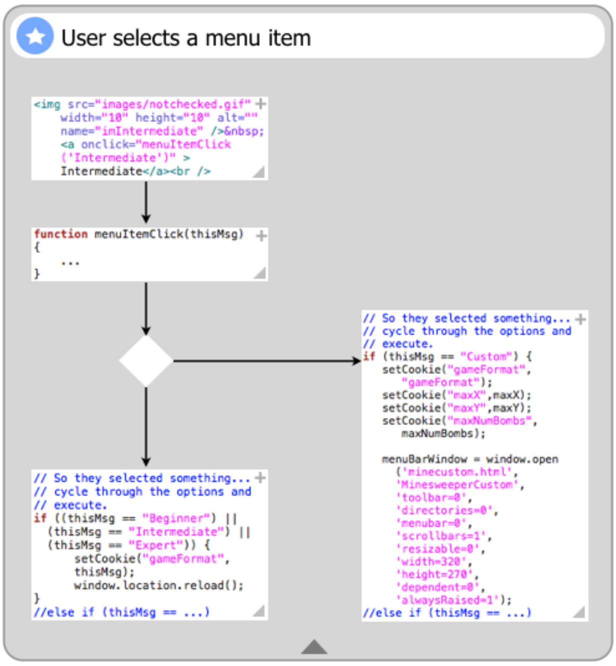
We take advatange of the various schemes that developers are familiar with.

Finally, since code modification inevitably requires developers to locate different code segments based on the tasks, another feature that we would also like to cover is to provide a different, more seamless method of navigating a code project. The current most popular method employed by developers, the textual search, is not too efficient, as it demands too much cognitive load on the developers' part just to comb through the search results in order to identify the one result that is actually relevant. Our proposed design should enable developers to quickly identify and access the relevant parts of a code project without ever losing grasp of the tasks at hand.

We would like to offer a more efficient method to navigate a code project as an alternative to the textual search.

1.2 Overview and Contributions

In chapter 2, we illustrate the typical progression of activity that a developer may follow while facing a mainte-



```
<img src="images/notchecked.gif"
    width="10" height="10" alt=""
    name="imIntermediate" />&nbsp;
    <a onclick="menuItemClick
    ('Intermediate')" >
    Intermediate</a><br />
```

```
function menuItemClick(thisMsg)
{
    ...
}
```

```
// So they selected something...
// cycle through the options and
// execute.
if ((thisMsg == "Beginner") ||
  (thisMsg == "Intermediate") ||
  (thisMsg == "Expert")) {
      setCookie("gameFormat",
      thisMsg);
      window.location.reload();
}
//else if (thisMsg == ...)
```

```
// So they selected something...
// cycle through the options and
// execute.
if (thisMsg == "Custom") {
   setCookie("gameFormat",
      "gameFormat");
   setCookie("maxX",maxX);
   setCookie("maxY",maxY);
   setCookie("maxNumBombs",
      maxNumBombs);

   menuBarWindow = window.open
     ('minecustom.html',
     'MinesweeperCustom',
     'toolbar=0',
     'directories=0',
     'menubar=0',
     'scrollbars=1',
     'resizable=0',
     'width=320',
     'height=270',
     'dependent=0',
     'alwaysRaised=1');
//else if (thisMsg == ...)
```

Figure 1.2: How a typical composition looks like in Code Mixer. The graph elements are the thumbnail images of the actual contents, and the graph structure is based on the user's whims.

Based on existing research, we identify the important areas that our work should focus on.

nance task, which is then followed by a set of problems that may prevent developers from finishing the task effectively. Meanwhile, chapter 3 lists the various tools and algorithms that have been proposed to help solve these problems. From these two chapters we identify the areas that we should focus on to make our eventual design as effective—and as attractive to developers—as possible.

We introduce the idea of *compositions* that represent developers' various problems.

Chapter 4 sees us proposing a novel approach to solving developers' problems by introducing the idea of *compositions* that represent the various problems that developers may face, regardless of their types. A composition can also represent the pieces of information that are shared among developers, making the process of exchanging solutions that much easier. We then formulate a set of design principles for a tool that supports such novel scheme.

The main contribution of this thesis work comes in chapter 5, where we elaborate on our incremental design phases that resulted in Code Mixer, a tool that provides assistance for developers throughout the entirety of the maintenance period. Code Mixer contains various compositions that offer a *task-oriented* outlook into a code project, allowing developers to focus only on the most relevant parts of source code at a time. It also puts emphasis on the *visual identities* of the compositions by allowing users to build their own *solution graphs* and utilizing content-based thumbnail images as the graph elements. Moreover, Code Mixer also offers a seamless method to navigate a code project, while simultaneously support lightweight distribution and reuse of compositions.

Code Mixer is a tool that offers help throughout the entirety of the maintenance period.

We investigate the effectiveness of our design in chapter 6 by conducting qualitative user studies. We build a simulated environment and present it to a group of users to get their overall sentiments of our design. Users' reactions are unanimously positive on the general idea of Code Mixer, although some design details receive a more mixed reaction. Ultimately, based on the result of the user studies, some final design adjustments, as well as some new features, are incorporated into our final design of Code Mixer.

User study participants reacted positively to the idea of Code Mixer.

Chapter 2

The Lives Of Developers

2.1 A Typical Work Session

Software maintenance typically occupies a big portion of a software lifecycle.

If you enter a room full of developers—furrowed brows, staring intently at the monitors with fingers furiously hitting the keyboards—it is a safe bet that the majority of these hard-working fellows are currently knee-deep in software maintenance. Another safe bet: these developers wish that the software they are tending to has a long, illustrious life, bringing plenty of benefits to its users. In order to ensure this, a big chunk of time must be devoted to maintaining said software, fixing bugs and adapting it to new contexts of use, a lengthy period that typically lasts for as long as the software is still used [Ko et al., 2006].

Developers spend the biggest share of their time on bug fixing.

Indeed, multiple studies, such as those by LaToza et al. [2006] and Singer et al. [1997], have shown that on average, developers spend around half of their time on bug fixing, an activity that is so closely related to software maintenance, they are often interchangeable. The rest of their working time, developers are either making enhancement to the system (e.g., adding new features) or making code more maintainable. However, it needs to be emphasized that this distribution of work time will depend greatly on the roles of developers, as well as how far along a software's lifecycle has elapsed. [LaToza et al., 2006].

The two most prominent activities that developers do are searching and looking at code.

Going deeper still into the work habit of developers, one could see that the two most prominent activities that they do are looking at code and performing a search in the code [Singer et al., 1997, Ko et al., 2006, Murphy et al., 2006]. In fact, Ko et al. [2006] found in their observations that in most cases, developers would begin a maintenance task with a textual search. Afterwards, they would begin navigating the relationships among the items they have found, which is often done by following a recurring, structured approach depending on work context. This owes to the fact the most maintenance tasks require making some changes to code (this includes adding new code lines), and developers simply wish to understand the code prior to making said changes [Roehm et al., 2012], an endeavor we would dub as *code comprehension*.

Definition: *Code Comprehension*

CODE COMPREHENSION:
A developer's endeavor to gain insight into the rationale and/or intention behind a piece of code or software system. This involves trying to make sense of implicit knowledge such as why a certain piece of code is written or structured the way it is, what it tries to accomplish, why it exists in the first place, and so forth.

Code comprehension often starts with exploring the code itself, and when this fails, developers turn to their colleagues.

Developers often try to conjure mental models of code.

During a code comprehension phase, developers normally rely on the code itself [LaToza et al., 2006], but reading source codes alone sometimes is not enough to provide them with enough information to proceed with their tasks. Singer et al. [1997] distributed a questionnaire in their study, and most developers claimed that they mostly consult the various code documentations during this troubled time. Curiously, various studies have observed otherwise: developers turn to their colleagues for help, barely putting any effort in finding the official documentations, let alone reading them [LaToza et al., 2006, Roehm et al., 2012]. This implies that the importance of design documents is more salient in theory than in practice. Moreover, LaToza et al. [2006] posited that developers also often try to conjure a *mental model* of code. A developer's mental model often contains the implicit knowledge behind a piece of code, and is typically created based on—and therefore limited to—the task at hand. Consequently, these mental models are largely isolated in nature, as opposed to being smaller parts of a bigger, whole picture. Developers use these mental models as a way to help themselves make sense of code, so different developers may come up with different mental models for the same piece of code.

Developers often employ the opportunistic programming approach, which also includes information foraging.

Once the developers feel they have gained enough insight into the code rationale, they will begin the actual modification of the code. More and more developers employ the *opportunistic programming* approach for this purpose, which is a programming practice that emphasizes speed and ease of development over code robustness and maintainability [Brandt et al., 2009]. This approach involves developers who are mostly concerned with finding and understanding the correct solution to a certain problem as quickly as possible, and they often do it by tailoring and mashing-up existing systems. Furthermore, as the Web gained more and more prevalence, developers resort to performing *information foraging* on the Web to find existing solutions that they can integrate into their own work. These web foraging activities may include anything from finding libraries and code examples to consulting online documentation, forums, and question-and-answer sites [Hartmann et al., 2011]. Ultimately, information foraging is popular because they serve multiple purposes: (1) as supports for *learning-by-doing* of new concepts, (2) as *clarifications* of existing knowledge, and (3) as *reminders* of typically low-level, syntactic information.

Definition: *Information foraging*

INFORMATION FORAGING:
Activities associated with assessing, seeking, and handling information sources. Such search will be adaptive to the extent that it makes optimal use of knowledge about expected info value and expected costs of accessing and extracting the relevant info [Pirolli and Card, 1995].

Developers also often employ copy-and-paste programming practice.

Another popular programming practice that is often utilized during an information foraging period is the *copy-and-paste* programming practice. It is mostly used when developers wish to integrate some information that they have gathered oppotunistically: they simply copy and paste dif-

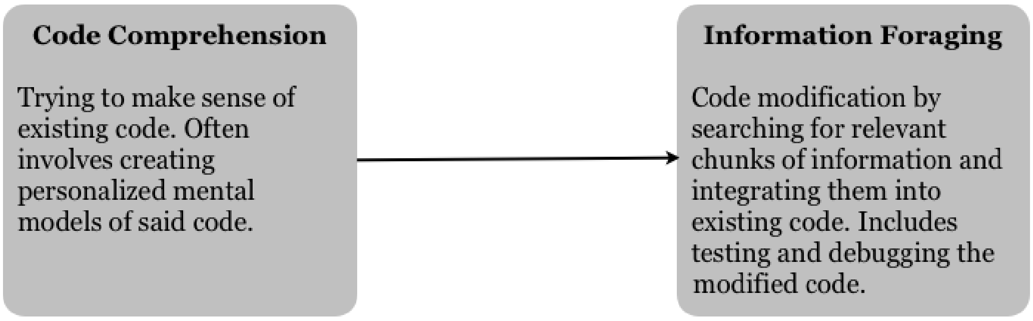

Code Comprehension
Trying to make sense of existing code. Often involves creating personalized mental models of said code.
Information Foraging
Code modification by searching for relevant chunks of information and integrating them into existing code. Includes testing and debugging the modified code.

Figure 2.1: A typical work progression followed by developers during a maintenance task.

ferent chunks of information into their own work. However, according to Kim et al. [2004], developers also use this programming practice for other uses, for instance to relocate, regroup or reorganize code (to make it more akin to the developer's mental model of said code), to reorder code fragments, or to refactor code manually. Kim et al. [2004] further proposed that the most prominent use of copy-and-paste is to reuse an existing code snippet as a structural template for another code snippet. This, of course, is closely related to the *reminder* function of information foraging.

Developers often delay testing code copied from the web.

Finally, developers will perform some debugging to verify the correctness of the changes that they have made. It has to be noted that developers often delay testing code copied from the web, making it harder to track errors [Brandt et al., 2009]. Furthermore, Roehm et al. [2012] discovered that developers rely heavily on compiler message and searching capabilities to deal with said errors. In addition, Brandt et al. noted that developers also go back to the web to *clarify* the appropriate output that a code should have.

Thus far we have tried to illustrate the progression that a developer may follow in a typical maintenance activity: they go from trying to understand a code, followed by implementing some changes with the help of opportunistic programming and copy-and-paste practices, to finally testing and debugging the changes that they have made. For the sake of brevity, these activities can roughly be divided into two groups: *code comprehension* and *information foraging*, with the former covering the first phase in the progression and the latter covering the rest. In the next section, we will discuss the multitude of problems that developers encounter along the way.

2.2 The Various Predicaments

In this section, we will take a more detailed look into some of the many stumbling blocks that prevent developers from having a smooth, productive work session. To make matters simpler, we follow the same narrative structure as from the previous section.

One of the biggest problem is understanding the rationale behind code.

We have established in the previous section that developers typically begin a maintenance task with a code comprehension phase, which involves trying to grasp the intention of the working code. The very process of code comprehension itself is actually one of the major problems that developers would have to overcome. As a matter of fact, LaToza et al. [2006] learned that 82% of the participants in their study agreed that it takes a lot of effort to understand "why the code is implemented the way it is." One can assume that it is only natural for a developer to experience some bumps while trying to acquire the reasonings behind a code implemented by another developer, but LaToza et al. [2006] further uncovered the disconcerting fact that some developers don't even remember the intention of the code that they themselves wrote!

Developers typically neglect code comprehension tools, relying on textual search instead.

It does seem rather curious that with the wealth of software comprehension tools available out there, understanding code remains one of the major hurdles to developers. Indeed, these tools (which includes software visualizations, concept location, or software metric tools) bear their own set of problems: according to a recent study by Roehm et al. [2012], developers simply choose not to use them. Instead, they opt for the humble textual search, which explains how it dominates the working time of developers during the code comprehension phase. While it is true that most of these tools have undergone multiple user studies which prove their worth, as Singer et al. [1997] posited, a user study environment has one glaring contrast to the real world: in the real world, users are free not to use these tools. And not use it they do.

A developer's work is often hindered by interruptions from others.

Another problem during the code comprehension phase progresses outwardly. Namely, the developers with the comprehension problems become the roots of another problem. We have mentioned that when failing to understand code on their own, developers would seek the guidance of their peers. By doing this, they inadvertently cause *interruptions* on said peers' work progression. As a matter of fact, LaToza et al. [2006] revealed that most developers feel that they are being interrupted too much. They find these interruptions bothersome, since when developers are interrupted, they switch tasks and often lose focus on their own task states before the interruption. Recovering from interruptions is a substantial problem and they even risk creating bugs if they remember incorrectly. However, Ko et al. [2006] further observed that developers are usually very careful to manage important tasks before acknowledging interruptions.

Developers often avoid real program comprehension, only focusing on the task at hand.

And yet, despite the strenuous, often costly measures taken to gather the logic behind a code, developers have been shown not to retain this invaluable information. The reasons are twofold. Firstly, developers tend to avoid real program comprehension, and are only concerned with the task at hand [Singer et al., 1997, Roehm et al., 2012]. Hence, the isolated nature of the mental models that they build around a code logic, and the inclination to lose them. Singer et al. dubbed this condition"Just In Time Comprehension" and noted that once they move on to another task, developers often forget the details that they have acquired and will have to re-explore parts of the system when they next encounter them.

Developers rarely record the knowledge that they have acquired.

The second reason behind low level of knowledge retention is the fact that developers simply choose not to record the information that they have collected. LaToza et al. [2006] suggested that some developers simply find it too much of a hassle to check in and out a code just to add some comments, and indeed, some feel these often abstract information are just not authoritative enough to warrant an official record. On the other hand, Roehm et al. [2012] remarked that some developers do utilize temporal notes on comprehension support, but these are used personally and not archived or reused. These valuable information will inevitably dissipate from memory, and other developers having the same problem will have to repeat the same ardurous process that should not have happened in the first place.

It can be hard to locate relevant information on the Web.

We will now proceed to the stage where developers are confident enough in their understanding of the code and are performing opportunistic programming to support task completion. In the very first step of this approach, the *web foraging*, already they stumble into some problems. This is due to the nature of the Web that serves as an enormous repository of information.While it offers convenience in the form of a wide breadth of choices, it can also hinder the process of locating the most relevant examples [Brandt et al., 2009]. Of course, the fact there is a very low probability of two people settling on the same terms to describe a condition will complicate the selection process even more [Furnas et al., 1987]. Furthermore, the sheer accessibility of the Web also contributes to the low information retention level of developers, since as Brandt et al. pointed out, developers have the tendency to treat it as an external memory aid. Consequently, developers often waste time roaming the web to find the solution to the same problem over and over again.

Integrating information gathered from the web can be challenging.

Once developers have decided on some information that they want to reuse, incorporating them into the working code is not always a straightforward task. A lot of refactoring may be required to update the code, particularly when the context of the original information is different from the developers' [Wightman et al., 2012]. The heavy-handed use of copy and paste practice does not help matter either, as this can lead to developers making unnecessary duplications of code snippets, increasing the possibily of bugs spreading around the system [Kim et al., 2004]. Finally, it is worth reiterating that the tendency to delay testing from a copied code from the Web can further exacerbate developers' ability to manage errors [Brandt et al., 2009]. Some

Code cloning may spread bugs around the system

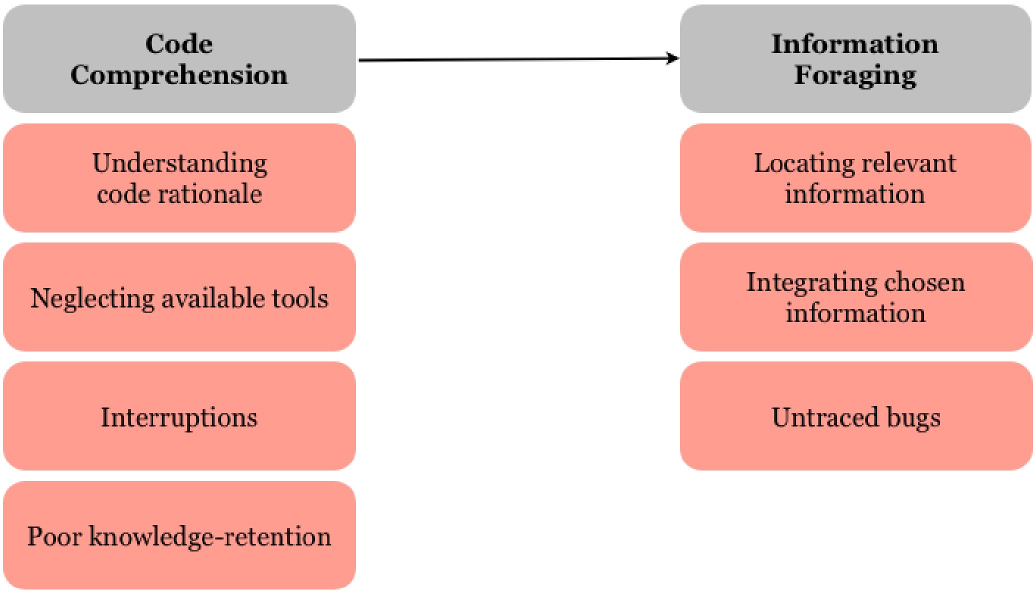

Code Comprehension
Understanding code rationale
Neglecting available tools
Interruptions
Poor knowledge-retention
Information Foraging
Locating relevant information
Integrating chosen information
Untraced bugs

Figure 2.2: The set of problems that developers typically face during software maintenance.

developers even assume that a code example would simply work and neglect to adapt portions of said example to fit their working code [Brandt et al., 2010]. Either way, the bugs introduced while borrowing a piece of code are often hard to find, and if developers happen to make multiple clones of this buggy code, they will need a lot of efforts to mitigate the damage.

Figure 2.2 depicts the set of problems that each maintenance period would entail. With the plethora of troubles lurking in every corner, ready to thrawt developers' every step from understanding to remembering to debugging, it is a wonder that developers can get any work done at all. Naturally, this is not an exclusive concern of ours, as many have tried to produce a variety of lifelines for developers to use in times of need. These tools will be analyzed in the next chapters.

Chapter 3

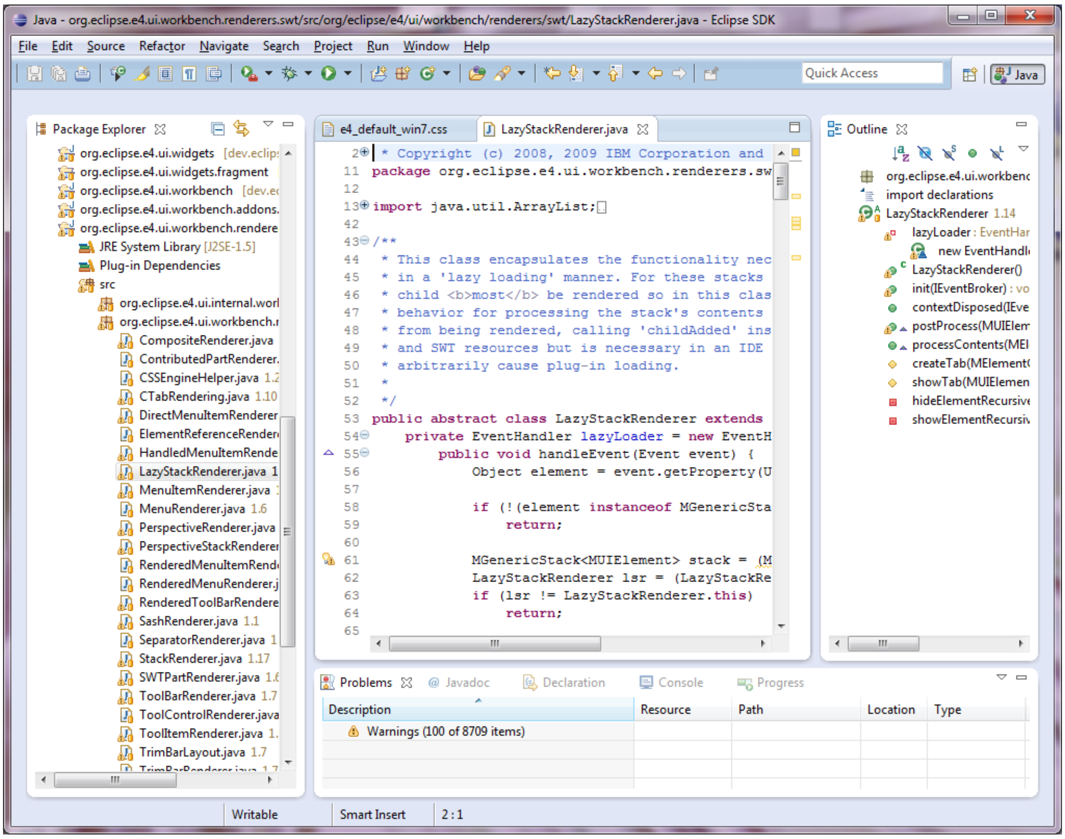
Related Work

Numerous amount of efforts have been put forth into making developers' lives easier, and we will discuss some of them in this chapter. Following the progression of activities that have been established in the previous chapter, we will start by focusing on the various researches that revolve around code comprehension, followed by those more focused in information foraging, and finally capping off with some tools that offer somewhat unconventional programming approaches.

Most modern IDEs uses the "bento box" approach in their interfaces.

In order to better understand how the tools presented in this chapter make a difference for developers, we would first establish the typical programming environment that most developers work in: the *Integrated Development Environment* (IDE). As its name suggests, an IDE combines many aspects of programming (e.g., editing, debugging, analyzing, and so forth) into one single interface. While each IDE has its own unique qualities that differentiate it from the others, most of them follow roughly the same user interface approach: the "bento box" [DeLine and Rowan, 2010]. These IDEs partition its interface into multiple "boxes," with each box corresponds to one particular functions, such as the code editor, the project navigator, the output display, and so on.

Most modern IDEs rely heavily on symbol cross-referencing and hyperlinks [DeLine and Rowan, 2010].



```
 * Copyright (c) 2008, 2009 IBM Corporation and
package org.eclipse.e4.ui.workbench.renderers.sw

import java.util.ArrayList;

/**
 * This class encapsulates the functionality nec
 * in a 'lazy loading' manner. For these stacks
 * child <b>most</b> be rendered so in this clas
 * behavior for processing the stack's contents
 * from being rendered, calling 'childAdded' ins
 * and SWT resources but is necessary in an IDE
 * arbitrarily cause plug-in loading.
 *
 */
public abstract class LazyStackRenderer extends
    private EventHandler lazyLoader = new EventH
        public void handleEvent(Event event) {
            Object element = event.getProperty(U

            if (!(element instanceof MGenericSta
                return;

            MGenericStack<MUIElement> stack = (M
            LazyStackRenderer lsr = (LazyStackRe
            if (lsr != LazyStackRenderer.this)
                return;

```

Description	Resource	Path	Location	Type
⚠ Warnings (100 of 8709 items)				

Figure 3.1: A sample interface of a conventional IDE. This Eclipse user interface is divided into multiple rectangular areas, each with a different purpose. Figure taken from Eclipse download page[a].

[a] http://download.eclipse.org/

The heavy reliance on symbol names may cause troubles for developers.

Consequently, when it comes to searching for a certain object, users typically start from the object's name, and then navigate through the search result using the "find next" feature that lets them jump across a code project to find the next instance of the submitted name until they find what they are looking for. Similarly, another popular feature, *code completion*, is often based on textual inputs as well. For instance, for method completion, the feature would only consider the receiver's declared type that has been typed in by the user, often resulting in an overwhelming number of suggestions for completions [Bruch et al., 2010]. On a related note, conventional search for code examples is typically rendered using techniques that were developed for standard textual documents [Bruch et al., 2010], which often fail to effectively capture source code's rich implicit knowledge and inherent structure. Ultimately, these are the various shortcomings that some of the tools presented in this chapter would directly address, while the rest are more concerned with providing a method that could enhance the more conventional programming practices.

3.1 Software Comprehension Tools

This area of research is geared towards improving code readability and accessibility. Despite the findings by Roehm et al. [2012] that developers often neglect to utilize this sort of tools (more details in 2.2 "The Various Predicaments"), they nevertheless have a lot of sound ideas that are highly relevant to the topic of this master thesis.

3.1.1 SHriMP Views

The SHriMP technique is concerned with presenting a large amount of information.

The SHriMP (Simple Hierarchical Multi-Perspective) visualizazion technique was designed to address the general challenge of presenting a large amount of information [Storey and Muller, 1995]. The researchers were particularly invested in finding the most effective way to filter and visualize information, so as to make it as easy to understand as possible. In the field of software engineering, this goal is closely related to the process of understanding code, particularly during reverse engineering.

Nested graphs provides concurrent views of multiple levels of abstractions.

SHriMP builds visualizations using a combination of *nested graphs* and *fisheye view*. Nested graphs, which uses the "graphs within graphs" approach, are responsible for detailing the structure and organization of a software system. On top of the more common depiction of software artifacts and their relationships, nested graphs can also illustrate composites and subsystems within a software, and thus they provide concurrent views of multiple levels of abstraction [Storey and Muller, 1995].

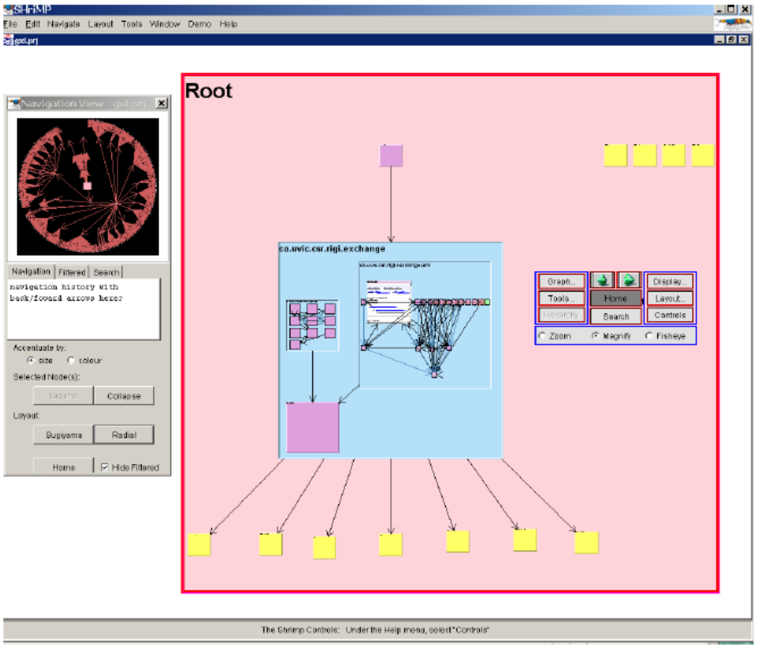

SHriMP
File Edit Navigate Layout Tools Window Demo Help
gxl.prj
Navigation View - gxl.prj
Navigation Filtered Search
navigation history with back/foward arrows here:
Accentuate by:
● size ○ colour
Selected Node(s):
Expand Collapse
Layout:
Sugiyama Radial
Home ☑ Hide Filtered
Root
ca.uvic.csr.rigi.exchange
Graph... Tools... Hierarchy
Home Search
Display... Layout... Controls
○ Zoom ● Magnify ○ Fisheye
The Shrimp Controls: Under the Help menu, select "Controls"

Figure 3.2: Visualization of a Java Project in SHriMP. Shown here is the root of the project in the form of a graph. To get a more detailed outlook, users can zoom in until the desired level is reached. Figure taken from Storey et al. [2002].

Fisheye view lets user focus on one specific part while maintaining the whole context.

Meanwhile, fisheye view prevents the resulting visualization from being too overwhelming. It works by letting users zooming in on a certain node while simultanously shrinking—but not hiding—the rest of the diagram. This way, a user can focus on the most relevant part while never losing track of how it is located in the context of the overall structure. Furthermore, users can zoom in all the way to the code level, and thus are given a seamless transition between implementation details and the more high-level outlooks of the system [Storey and Muller, 1995]. We would eventually incorporate this navigation style into our design, the details of which will be revealed in the later part of this paper.

While SHriMP views offer some advantages in its ability to offer a quick glance at the structural aspects of a software along with an intuitive approach to navigating, it can also become a hindrance. For instance, in the case where a developer is only interested in one small part of a very large system, keeping all the other unneccesary elements on the screen may prove to be more harmful than useful [Storey and Muller, 1995]. Consequently, a SHriMP view may work best when combined with the other more conventional views.

SHriMP views should be combined with more conventional views

3.1.2 Relo

Sinha et al. [2006] introduced Relo as a direct answer to the developers' problem of building their own mental models of existing, unfamiliar code. It is particularly aimed at addressing the complexity that comes with the various abstractions introduced by modern programming paradigms such as object-oriented programming and design patterns. These abstractions and their interactions with one another force a developer to explicitly keep track of their context while exploring and trying to understand the rationale behind code. For instance, when investigating a method of a class, a developer has to check for inheritance and whether said method overrides a method of a parent class or not.

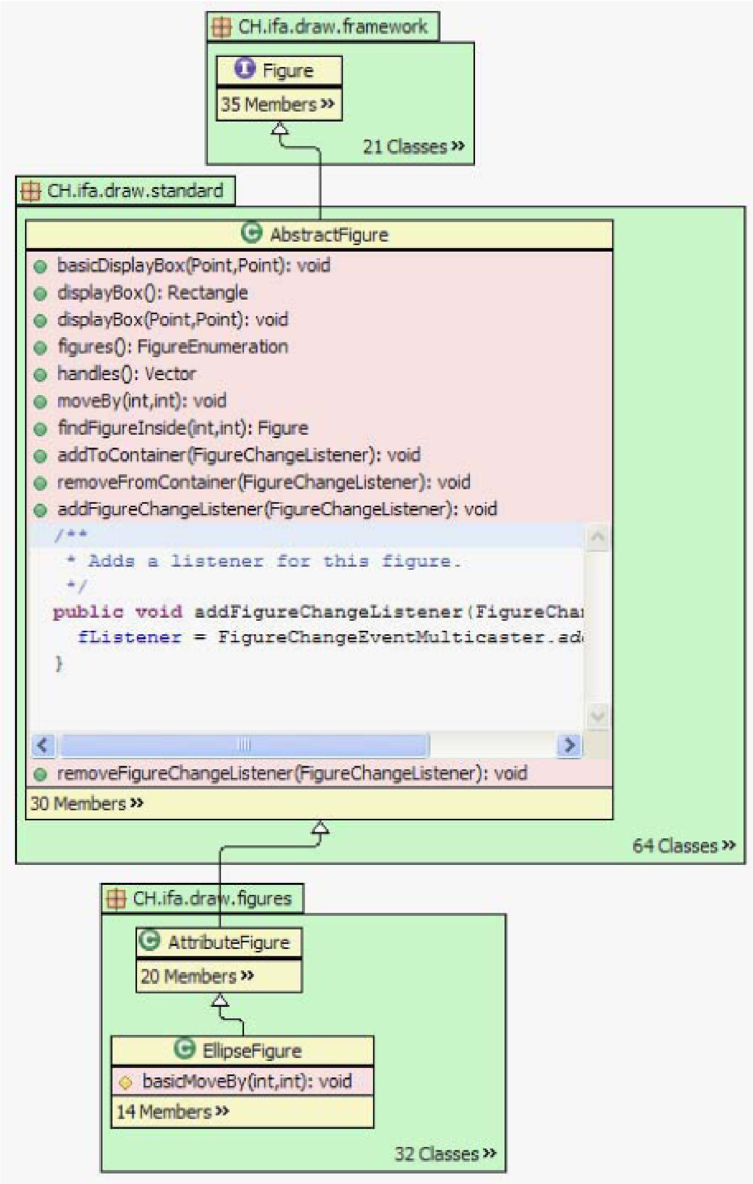
Relo deals with the various abstractions that come with modern programming paradigms.

Relo works by letting developers select small, relevant parts of large code bases. It provides an *interactive exploration* tool that lets users select, add and remove code parts from the working space, allowing them build and trim a diagram of relevant information as they go. Based on the assumption that developers often prefer to focus on small parts that they need rather than look at the whole picture [Singer et al., 1997, Roehm et al., 2012], a Relo work space always starts with a single code element such as a package or a method, from which a developer can interactively build a bigger, more complex structure as needed.

Relo lets user interactively explore relevant parts of a system

The tool then automatically lays out a *graphical representation* of selected code fragments, following a layout rules that position elements based on their relationships. For instance, a vertical layout represents inheritance hierarchies, a left-to-right positioning represents call stacks, while a container layout denotes package and class containment.

Relo generates graphical representation of the selected code fragments


CH.ifa.draw.framework
Figure
35 Members »
21 Classes »
CH.ifa.draw.standard
AbstractFigure
basicDisplayBox(Point,Point): void
displayBox(): Rectangle
displayBox(Point,Point): void
figures(): FigureEnumeration
handles(): Vector
moveBy(int,int): void
findFigureInside(int,int): Figure
addToContainer(FigureChangeListener): void
removeFromContainer(FigureChangeListener): void
addFigureChangeListener(FigureChangeListener): void
removeFigureChangeListener(FigureChangeListener): void
30 Members »
64 Classes »
CH.ifa.draw.figures
AttributeFigure
20 Members »
EllipseFigure
basicMoveBy(int,int): void
14 Members »
32 Classes »

```
/**
 * Adds a listener for this figure.
 */
public void addFigureChangeListener(FigureCha
  fListener = FigureChangeEventMulticaster.ad
}
```

Figure 3.3: A sample Relo view that shows how a code editor is embedded in the diagram. Users can modify code directly from this embedded editor. Figure taken from Sinha et al. [2006].

Another interesting feature is the ability to zoom in to view and directly edit code from inside a diagram, using embedded text editors, as depicted in Figure 3.3.

Relo may be more beneficial for large code bases.

Relo has been evaluated against a sizeable code base consisting of around 150,000 lines of code, and the results have been quite positive. Participants in the study generally agreed that Relo has a low learning curve and can especially help them in navigating large code bases. However, when developers are working with a small amount of code artifacts (three or less), they felt that working with Relo was a bit cumbersome, since Relo's mouse-based navigation style does not offer as many shortcuts as keyboard-based navigations [Sinha et al., 2006].

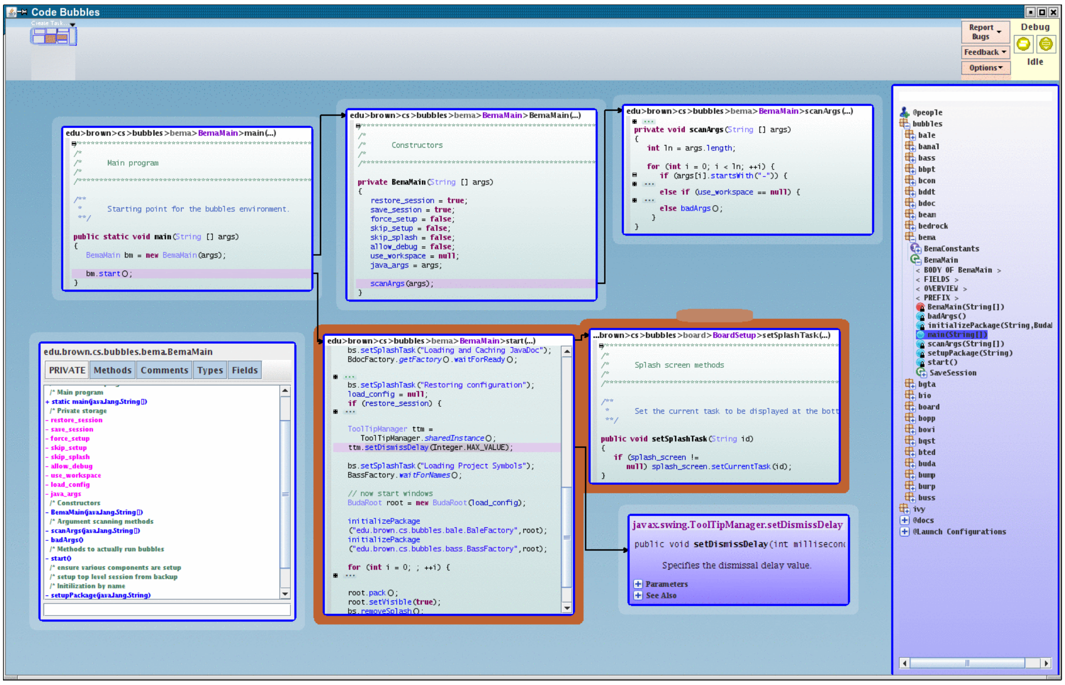
3.1.3 Code Bubbles

Code Bubbles tries to eliminate isolated views of code fragments

Code Bubbles [Bragdon et al., 2010] is an approach to code viewing that is based on small, editable fragments of codes called "bubbles." This tool originated from the realization of how much time is spent by developers navigating the file-based environments of most IDEs, particularly by opening and repositioning windows or jumping back and forth across multiple tabs. The researchers believe that this isolation of relevant fragments of code may overload developers' working memory and hinder them in their quests to understand code intentions. Therefore, they set to provide a concurrent viewing of code fragments that would enable developers to form a working set that can be used to inspect the relationships of several code chunks. They believe that such tool will let developers navigate unfamiliar code without "getting lost," since a simple glance would be enough to refresh their context.

A bubble contains a code fragment and is fully editable.

Code Bubbles is based on the metaphor of, naturally, a bubble. A bubble is a fully editable and interactive view of a code fragments whose scope may extend from a small group of variables to full functions. In contrast to typical desktop windows, a bubble boasts minimal border decoration, avoids content clipping through the use of automatic reflow, and prevents overlapping other bubbles by push-



edu>brown>cs>bubbles>bema>BemaMain>main(...)

```
/************************************************************
/*
/*      Main program
/*
/************************************************************

/**
 *      Starting point for the bubbles environment.
 **/

public static void main(String [] args)
{
   BemaMain bm = new BemaMain(args);

   bm.start();
}
```

edu>brown>cs>bubbles>bema>BemaMain>BemaMain(...)

```
/************************************************************
/*
/*      Constructors
/*
/************************************************************

private BemaMain(String [] args)
{
   restore_session = true;
   save_session = true;
   force_setup = false;
   skip_setup = false;
   skip_splash = false;
   allow_debug = false;
   use_workspace = null;
   java_args = args;

   scanArgs(args);
}
```

edu>brown>cs>bubbles>bema>BemaMain>scanArgs(...)

```
...
private void scanArgs(String [] args)
{
   int ln = args.length;

   for (int i = 0; i < ln; ++i) {
      if (args[i].startsWith("-")) {
...
      else if (use_workspace == null) {
...
      else badArgs();
    }
}
```

edu.brown.cs.bubbles.bema.BemaMain

```
 /* Main program
+ static main(java.lang.String[])
 /* Private storage
- restore_session
- save_session
- force_setup
- skip_setup
- skip_splash
- allow_debug
- use_workspace
- load_config
- java_args
 /* Constructors
- BemaMain(java.lang.String[])
 /* Argument scanning methods
- scanArgs(java.lang.String[])
- badArgs()
 /* Methods to actually run bubbles
- start()
 /* ensure various components are setup
 /* setup top level session from backup
 /* Initilization by name
- setupPackage(java.lang.String)
```

edu>brown>cs>bubbles>bema>BemaMain>start(...)

```
   bs.setSplashTask("Loading and Caching JavaDoc");
   BdocFactory.getFactory().waitForReady();

...
   bs.setSplashTask("Restoring configuration");
   load_config = null;
   if (restore_session) {
...

   ToolTipManager ttm =
      ToolTipManager.sharedInstance();
   ttm.setDismissDelay(Integer.MAX_VALUE);

   bs.setSplashTask("Loading Project Symbols");
   BassFactory.waitForNames();

   // now start windows
   BudaRoot root = new BudaRoot(load_config);

   initializePackage
   ("edu.brown.cs.bubbles.bale.BaleFactory",root);
   initializePackage
   ("edu.brown.cs.bubbles.bass.BassFactory",root);

   for (int i = 0; ; ++i) {
...

   root.pack();
   root.setVisible(true);
   bs.removeSplash();
```

...brown>cs>bubbles>board>BoardSetup>setSplashTask(...)

```
/************************************************************
/*
/*      Splash screen methods
/*
/************************************************************

/**
 *      Set the current task to be displayed at the bott
 **/

public void setSplashTask(String id)
{
   if (splash_screen !=
      null) splash_screen.setCurrentTask(id);
}
```

javax.swing.ToolTipManager.setDismissDelay

```
public void setDismissDelay(int milliseconc
```

Specifies the dismissal delay value.

Figure 3.4: A typical Code Bubble surface contains "bubbles" that contain parts of code. When users open the definition of a certain object, it is contained in another bubble, with an arrow indicating their connection. Bubbles with the same color belong to the same group. Figure taken from the Code Bubbles Home Page[a].

[a] http://cs.brown.edu/ spr/codebubbles/

ing each other out of the way. A set of bubbles co-exist in a virtual, pannable space, where together they serve as a working set of relevant information for developers [Bragdon et al., 2010].

Code Bubbles can improve the accuracy and speed of code comprehension tasks.

Code Bubbles has undergone both quantitative and qualitative evaluations with promising results. When compared to regular Eclipse controls during code comprehension tasks, participants with Code Bubbles outperformed those in the control group, both in terms of task completion and the time required to complete said tasks. Moreover, the results suggest that the performance improvement may not be solely attributed to the reduction in navigation time, which indicates that there is further cognitive benefits in a concurrent viewing approach. These results are further supported by the qualitative study, where the majority of developers reach positively to the tool, believing that Code

Bubbles can help them offload memory and "querying" code to answer specific questions, among other things.

3.1.4 Code Canvas

As with Code Bubbles, Code Canvas [DeLine and Rowan, 2010] was introduced as a result of dissatisfaction towards the visual representations of most popular IDEs. Said representations typically revolve around the "bento box" metaphor, where the screen is partitioned into special areas, each with its own specific purpose, such as code editors, class viewers, output message viewers, and so forth. While most developers have grown accustomed to following this approach, it does not erase the many problems it could cause. The frequent use of hyperlinks to jump around the project, for instance, could cause developers to "get lost in the code." Additionally, it may hinder the process of synthesizing relevant information scattered across a project to answer a specific question about a code.

Code Canvas is an alternative to the popular "bento box" approach of most popular IDEs.

Code Canvas then takes advantages of the advanced technical capabilities of most modern monitors to replace the segmentations of a bento box with a single, infitinely zoomable surface called a canvas. A canvas houses all of a project's documents, including source code, user interface designs, images, and so on. Each elements in the canvas is editable, so users simply zoom in into a desirable area to perform editing and zoom back out to get an overview of the project [DeLine and Rowan, 2010].

Code Canvas uses a single, infinitely zoomable surface.

Users are also allowed to modify the size and position of many of the elements in the canvas. This way, Code Canvas exploits users' own spatial memory to help them orient themselves in a project. This particular feature would actually make up one of our design properties that are going to be presented in the next chapter. Additionally, the canvas also serves as a visualization surfaces which displays multiple layers of project information including search result, test coverage, and execution traces [DeLine and Rowan, 2010].

Code Canvas exploits users' own spatial memory.

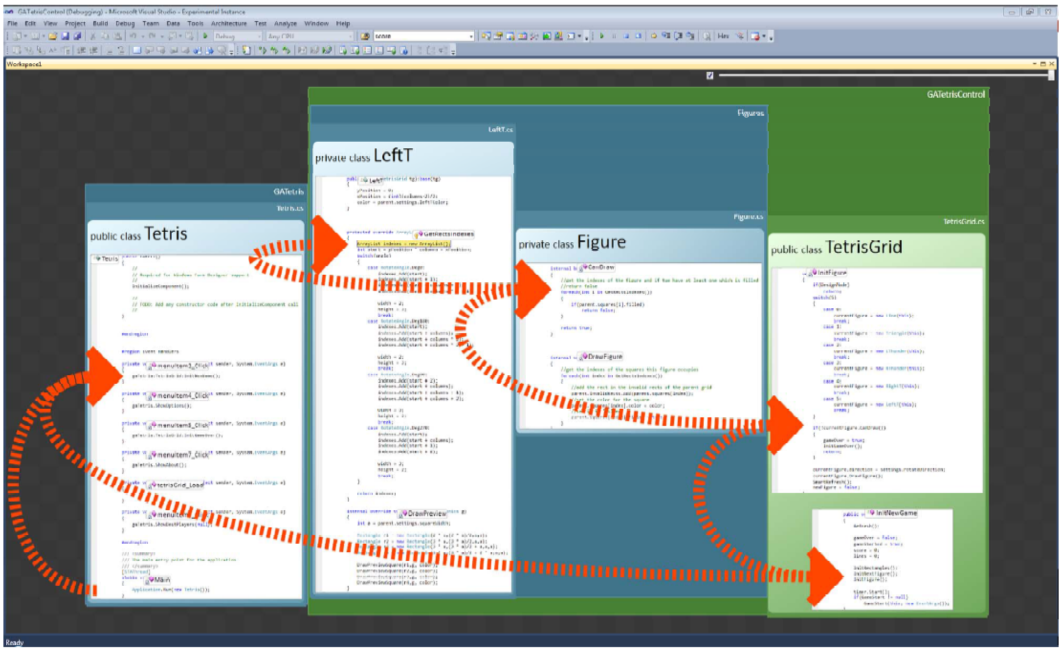

GATetrisControl (Debugging) - Microsoft Visual Studio - Experimental Instance
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
Workspace1
GATetrisControl
Figures
LeftT.cs
private class LeftT
GATetris
Tetris.cs
public class Tetris
Figure.cs
private class Figure
TetrisGrid.cs
public class TetrisGrid
Ready

Figure 3.5: A filtered view of Code Canvas, showing only the elements that are relevant to the current call stacks. Figure taken from DeLine and Rowan [2010].

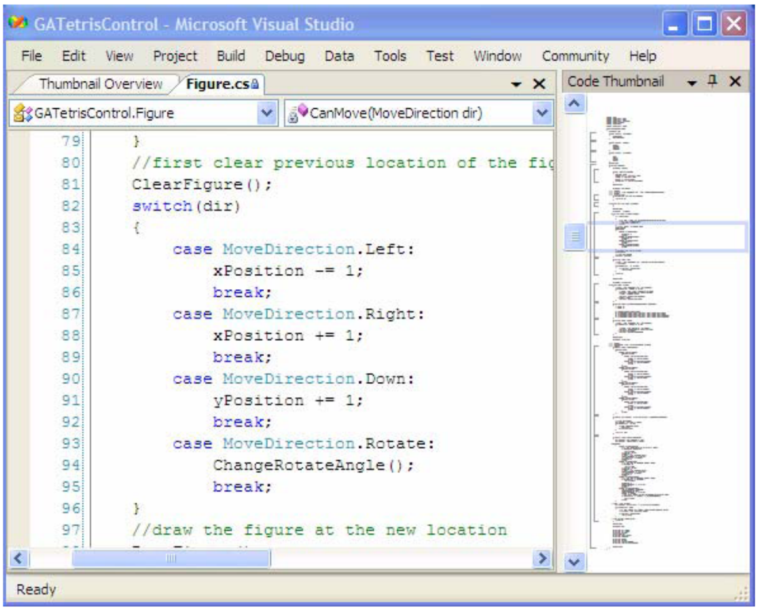
Debugger Canvas was developed to test the Code Bubbles and Code Canvas paradigms.

The team of Code Bubbles and Code Canvas collaborated to create an extension of Visual Studio that combines the two paradigms, called *Debugger Canvas* [DeLine et al., 2012]. User feedbacks have been quite positive, and the consensus was that Debugger Canvas is particularly beneficial for large projects such as those that involve long or complex code paths, or dynamically linked code, factories, or other indirect forms of control flow. However, the study also noted that the canvas idea is best deployed as a complement to the existing user experience, as opposed to a standalone substitute.

3.1.5 Code Thumbnails

Code Thumbnails were developed to reduce the visual uniformity of code.

DeLine et al. [2006] came up with Code Thumbnail as an alternative to the more conventional method of code navigating that relies heavily on texts, particularly symbol names. For instance, selecting a method requires knowing the name of the method, the name of the containing


GATetrisControl - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Test Window Community Help
Thumbnail Overview
Figure.cs
GATetrisControl.Figure
CanMove(MoveDirection dir)
Code Thumbnail
Ready

```
    }
    //first clear previous location of the fi
    ClearFigure();
    switch(dir)
    {
        case MoveDirection.Left:
            xPosition -= 1;
            break;
        case MoveDirection.Right:
            xPosition += 1;
            break;
        case MoveDirection.Down:
            yPosition += 1;
            break;
        case MoveDirection.Rotate:
            ChangeRotateAngle();
            break;
    }
    //draw the figure at the new location
```

Figure 3.6: A Code Thumbnail Scrollbar. Selecting a part of the scrollbar would make the main editor focus on the selected area. Figure taken from DeLine et al. [2006].

class and the class's containing namespace. This plethora of names could easily overwhelm developers, especially in large projects. Another problem that it tries to address is the visual uniformity of source code, which is a direct consenquence of the practice of "writing" code. Visual uniformity may lead to the cases in which developers waste their time by wandering back to a certain part of code over and over again, not realizing that it is the same part that they have already browsed through.

CT Scrollbar enhaces the vertical scrollbar in a single file.

Code Thumbnail consists of two user interface features in Microsoft Visual Studio: the Code Thumbnail Srollbar (CT Scrollbar) and the Code Thumbnail Desktop (CT Desktop). The CT Scrollbar is aimed at enhancing the typical vertical scrollbar for navigating within a single file. The idea is to embed a thumbnail image of the whole document to the scrollbar. The main purpose of this feature is for developers to use the shapes of the texts as visual landmarks without actually reading them [DeLine et al., 2006]. This is why the font size inside the CT Scrollbar is kept below the threshold

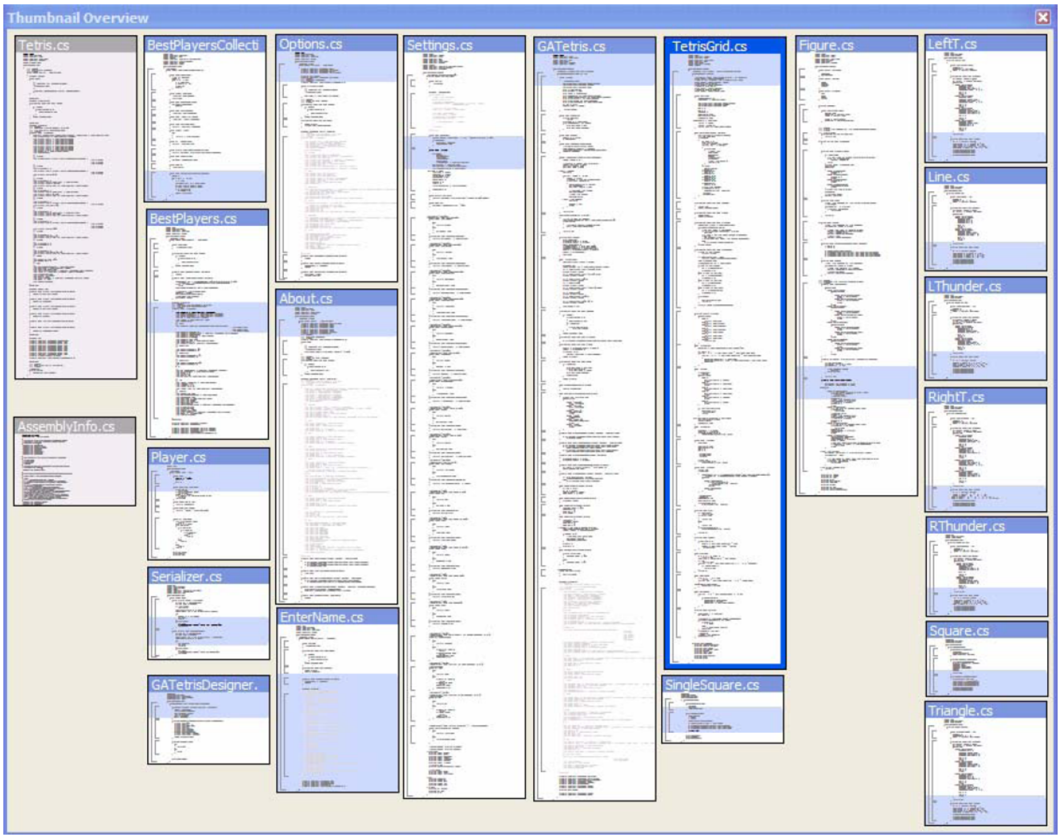


Figure 3.7: A Code Thumbnail Desktop. The grey background denotes a closed file. User can open a file directly by double clicking a thumbnail. Figure taken from DeLine et al. [2006].

of readability. On top of the regular scrolling, users can also click on a certain area in the CT Scrollbar to jump into its corresponding code in the main editor.

CT Desktop offers navigation among multiple files.

In the mean time, CT Desktop is to be used to navigate among multiple files. It displays the thumbnail image of every source files in the project, neatly arranged in a desktop-like surface. Each thumbnail is also equipped with a title bar that serves both as an identifier and an anchor for dragging. Just like in a desktop, users can freely rearrange the items as they see fit. Furthermore, Code Thumbnail is integrated with the standard search tools, so search results would be highlighted in both versions of the tool [DeLine et al., 2006].

Initial evaluations on Code Thumbnails yielded positive results. Quantitative analysis proved that it can increase speed of both method and file searching tasks. Furthermore, there is also a consensus among participants regarding its ease of use, learnability and preference over exising user interface. Accordingly, we found the method of utilizing content as a visual cue to be quite promising, and we are going to play around with it in the later stage of our design phase.

Code Thumbnails can improve searching taks.

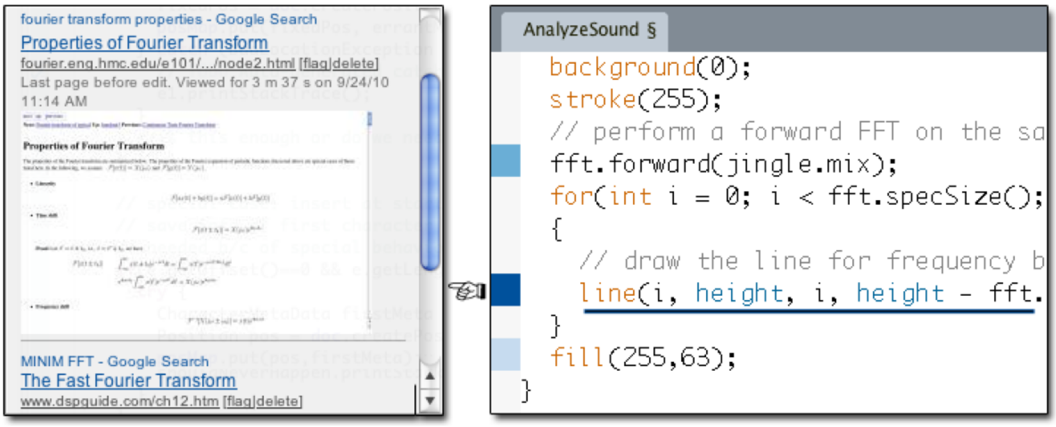
3.1.6 HyperSource

Despite the considerable amount of time spent on the Web to find useful information, developers rarely record the sources of the knowledge they sample. Once a code snippet gathered from the Web is deployed in the existing code, the connection to its source is lost. Consequently, the design decisions behind the borrowed snippet are obscured, often leaving those trying to grasp said decisions at a lost. Hartmann et al. [2011] proposed HyperSource as a bridge between a snippet and the browsing history traversed to get to said snippet. The advantages of said bridge is twofold: (1) it can help developers re-establish context on their own code and shorten the time wasted on re-retrieving previously used resources; (2) it brings code newcomers up to speed on design decisions and therefore enable them to become productive faster.

HyperSource bridges source code and browsing history.

Hypersource enhances conventional IDEs by creating associations between source code edits and browsing history [Hartmann et al., 2011]. A line of code is associated with a set of web pages visited just before a change is made on it. This association is denoted by a "line hightlight," which when clicked will reveal a panel that details the browsing history associated with the current line (See Figure 3.8). The browsing history contains information such as the links to the Web pages, the amount of time spent on each page, and thumbnail images of the pages. Users can then interact with these links directly from the panel, instead of having to open a separate Web browser.

User interact with browsing history by clicking "line highlights."



fourier transform properties - Google Search
Properties of Fourier Transform
fourier.eng.hmc.edu/e101/.../node2.html [flag|delete]
Last page before edit. Viewed for 3 m 37 s on 9/24/10 11:14 AM

Properties of Fourier Transform

MINIM FFT - Google Search
The Fast Fourier Transform
www.dspguide.com/ch12.htm [flag|delete]

```
  background(0);
  stroke(255);
  // perform a forward FFT on the sa
  fft.forward(jingle.mix);
  for(int i = 0; i < fft.specSize();
  {
    // draw the line for frequency b
    line(i, height, i, height - fft.
  }
  fill(255,63);
}
```

Figure 3.8: In Hypersource, line highlights denote the associations between code lines and browsing history. Figure taken from Hartmann et al. [2011].

HyperSource provides heuristic and manual filtering.

Additionally, HyperSource performs heuristic filtering on the set of links to weed out irrelevant links [Hartmann et al., 2011]. A page is considered relevant if it contains a copied text to the source code, is explicitly bookmarked by developers in the browser, or is the final page visited before editing. On top of this automatic filtering, Hypersource also allows users to perform manual filtering by allowing them to define a blacklist of domain names, as well as to clear all or individual items from the list.

More extensive study still required to gauge real benefits.

Informal evaluations revealed that HyperSource did help users understand unfamiliar code. Moreover, a separate study that required users to finish programming tasks using HyperSource resulted in relatively low number of line highlights, and therefore these highlights may also be used to indicate "interesting" sections of code. This observation supports the finding which states that during code comprehension tasks, users rarely focus on the whole code project, but instead opt to only pay attention on smaller, isolated code sections that are related to their tasks [Singer et al., 1997, Roehm et al., 2012].

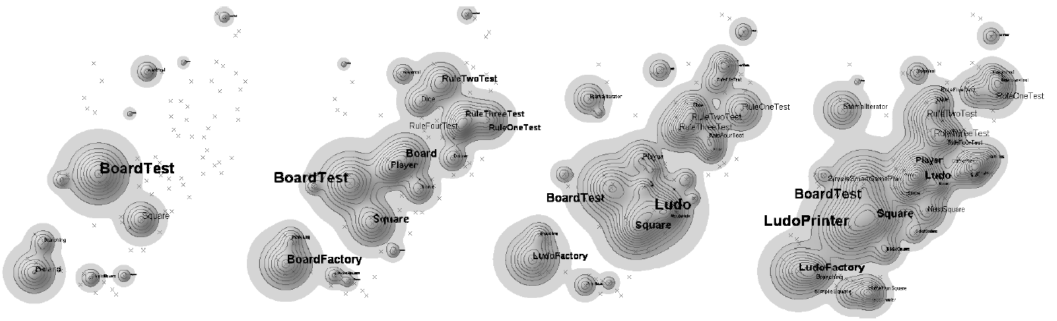


Figure 3.9: A set of software maps of an evolving system. Notice the relative stability of the vocabulary. Figure taken from Kuhn et al. [2008].

3.1.7 Software Cartography

Software Cartography by Kuhn et al. [2008] offered a different visualization approach than what have been presented thus far. It separates itself from the process of reading code, and instead is more concerned with presenting a high-level view of a software system. Software cartography tries to remedy the tendency of most visualization tools to present structures in arbitrary layouts. Thus, the main goal of this project is to come up with a consistent layout for software systems, not unlike those typically found in thematic atlas that present different phenomena such as population density to industry sectors using the same consistent layout.

Software Cartography is tailored to provide a consistent layout to display software systems.

The biggest hurdle to this approach is the fact that software systems do not possess tangible physical locations [Kuhn et al., 2008]. As a work-around, Software Cartography employs *vocabulary*, due to the assumption that vocabularies can effectively mask the technical details of code by identifying the key domain concepts of said code. Moreover, when a software expands and changes its structure, the underlying vocabulary tends to remain the same. The resulting visualizations, called *software maps*, can provide both a consistent layout for different kinds of thematic maps, as well as serve as comparisons for different stages of an evolving system, as can be seen in Figure 3.9.

Software Cartography uses vocabulary to substitute for physical locations.

To achieve this capability, Software Cartography employs the popular information retrieval technique, Latent Seman-

Software Cartography uses Latent Semantic Indexing and Multidimensional Scaling.

tic Indexing (LSI), to acquire *lexical similarity*, which illustrates how closely related the vocabularies among multiple software artifacts. The more similar the vocabularies between two software artifacts, the closer they are conceptually and topically. Additionally, Multidimensional Scaling is utilized to map the resulting vector spaces gleaned from LSI into the target visualization space.

Definition: *Latent Semantic Indexing (LSI)*

LATENT SEMANTIC INDEXING (LSI):
An indexing and retrieval method that aims to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. It was initially designed for information retrieval in search engines.

Definition: *Multidimensional Scaling (MDS)*

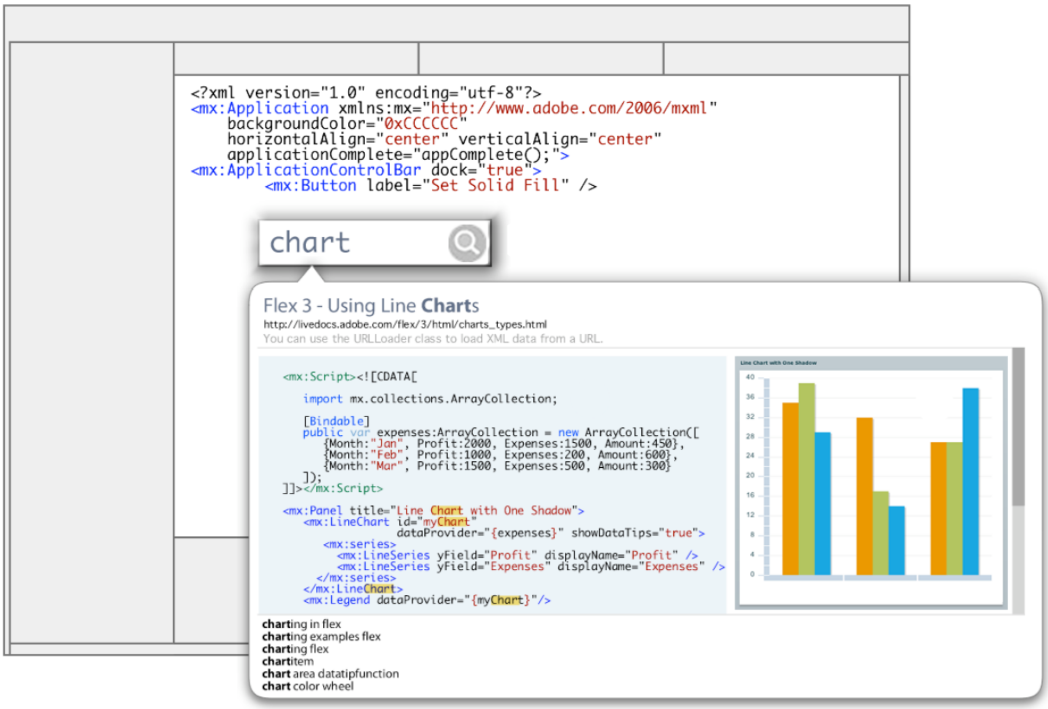
MULTIDIMENSIONAL SCALING (MDS):
A technique to map objects from m-dimensional space to two dimensions that tries to minimize a stress function while iteratively placing elements into a low-level space. MDS yields the best approximation of a vector space's orientation, i.e., preserves the relation between elements as best as possible.

3.2 Information Foraging and Code Reuse Tools

The tools in this section were designed to assist developers search for and locate relevant information for their code manipulation tasks. As we have mentioned in the previous chapter, information foraging is often followed by code snippet reuse (through copy & paste). Accordingly, some of the tools listed here support both endeavors.

3.2.1 Blueprint

It has become increasingly clear that developers are incorporating more examples from the Web into their code



```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="0xCCCCCC"
    horizontalAlign="center" verticalAlign="center"
    applicationComplete="appComplete();">
<mx:ApplicationControlBar dock="true">
        <mx:Button label="Set Solid Fill" />
```

chart

Flex 3 - Using Line **Chart**s

http://livedocs.adobe.com/flex/3/html/charts_types.html

You can use the URLLoader class to load XML data from a URL.

```
<mx:Script><![CDATA[

    import mx.collections.ArrayCollection;

    [Bindable]
    public var expenses:ArrayCollection = new ArrayCollection([
       {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
       {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
       {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
    ]);
]]></mx:Script>

<mx:Panel title="Line Chart with One Shadow">
    <mx:LineChart id="myChart"
                  dataProvider="{expenses}" showDataTips="true">
      <mx:series>
        <mx:LineSeries yField="Profit" displayName="Profit" />
        <mx:LineSeries yField="Expenses" displayName="Expenses" />
     </mx:series>
    </mx:LineChart>
    <mx:Legend dataProvider="{myChart}"/>
```

charting in flex
charting examples flex
charting flex
chartitem
chart area datatipfunction
chart color wheel

Figure 3.10: User performs search right from the IDE. Blueprint then augments search results with context. Figure taken from Brandt et al. [2010].

[Brandt et al., 2009]. However, most popular development environments provide little support for this example-centric approach to programming. As far as the IDE is concerned, everything is either written by developers themselves or imported modules. As a result, the connection between borrowed code and its source is often lost, making it difficult to reaccess the original source to verify design decisions. Furthermore, if the original reused code is later updated (e.g., to fix bugs), the developers have no way of knowing [Brandt et al., 2010].

Web search tools are typically separated from code editing tools.

Blueprint was developed as a way to integrate Web search into the development environment (in this case, Adobe Flex Builder)[Brandt et al., 2010]. As opposed to opening a new browser window, user can access the search feature from inside the IDE by using a hotkey. The most interesting part of Blueprint, however, is the way it favors an example-centric view of the search results. It augments the more conventional search results by providing context in the form of tex-

Blueprint lets user perform search from inside the IDE.

Blueprint automatically adds the source address in the comment.

tual descriptions, example code, and, if available, a running example. Once user decides to reuse and paste a code snippet into their own work, Blueprint automatically creates a comment in the code containing the web address and access time of said code, so the connection to the original source is maintained. This feature implies that Blueprint supports the *reminding* and *clarification* purposes of information foraging (See Chapter 2 "The Lives Of Developers").

Blueprint's search result view is not as effective as it can be.

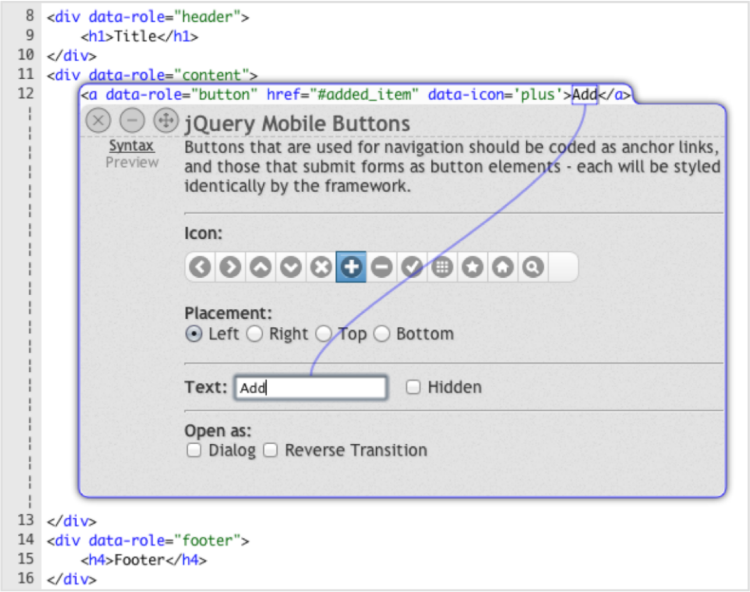
User studies suggested that Blueprint can indeed improve the efficiency of code example reuse. Participants with exposure to Blueprint started pasting code examples much faster than the other group, which ultimately would result in a faster overall time in finishing the tasks [Brandt et al., 2010]. However, the researchers themselves feel that the tool still does not display searh results in an efficient manner. For example, search terms need to be highlighted in the search result to make them easier to identify. Furthermore, users can only view one example at a time, rendering comparisons between multiple examples a little tricky.

3.2.2 Codelet

Most code examples lose their original contexts after integration.

Codelet, by Oney and Brandt [2012], is another tool that specialized in improving developers' interactions with code examples gleaned from external sources. Regardless of their sources (e.g., online forums, tutorial, etc.), code examples typically contain contextual information describing how the code works. Once they enter the user's code, however, they lose this rich, instructive information, and become indistinguishable from existing code. This may pose a problem, since a developer's interaction with a code snippet begins precisely at this point. In most cases, a code snippet would require some configurations before it can fit with the rest of the code, and a developer is therefore forced to consult with the original source in a separate window. Once the task is done, the snippet will be fully detached from its original context, gradually dissipating from developer's memory [Singer et al., 1997, Oney and Brandt, 2012].

To remedy this situation, Codelet was designed as a region



```
<div data-role="header">
    <h1>Title</h1>
</div>
<div data-role="content">
  <a data-role="button" href="#added_item" data-icon='plus'>Add</a>
```

jQuery Mobile Buttons

Buttons that are used for navigation should be coded as anchor links, and those that submit forms as button elements - each will be styled identically by the framework.

Icon:

Placement:
● Left ○ Right ○ Top ○ Bottom

Text: Add □ Hidden

Open as:
□ Dialog □ Reverse Transition

```
</div>
<div data-role="footer">
    <h4>Footer</h4>
</div>
```

Figure 3.11: A sample Codelet view containing textual descriptions and multiple parameters. Figure taken from Oney and Brandt [2012].

in the user's code that contains a block of example code and an associated interactive helper. The helper is a widget that helps user understand a code snippet by providing contextual texts and structured parameterizations (See Figure 3.11). In other words, developers can modify a code snippet without actually writing any code. Indeed, the helper can also inspect user's entire code base and tailor the code snippet based on features in the rest of the code. Furthermore, since a Codelet remains accessible even after configuration and integration is done, its context is never lost [Oney and Brandt, 2012].

A Codelet contains a block of example code and an interactive helper.

In order for Codelet to be truly useful, however, there should be a wide pool of code snippets to choose from. For this purpose, Codelets API was developed. This API is intended for those who specialize in authoring code libraries and documentations, so that the resulting Codelets can be as effective as possible, but all Codelets users can also build code examples if they wish to do so. The API is kept as simple as possible, so creating a Codelet would

Codelet API is available to build Codelets from scratch.

only require approximately the same amount of effort as creating its Web page version [Oney and Brandt, 2012].

Codelet can speed code reuse tasks, but is limited to one contiguous block of code.

Codelet has been shown to reduce both the amount of time spent and number of page refreshes during code example reuse tasks. During user studies, Codelet also increased participants' confidence in rewriting their code with no documentation by a moderate margin (4.63 vs 3.80 on a nine-point Likert Scale), and slightly improved their ability to understand the code that they had written (7.88 vs 7.80) [Oney and Brandt, 2012]. It was also noted, however, that the latest version of Codelet could only support code examples that consist of one contiguous block of lines, which is somewhat restrictive, since a lot of examples on the Web consist of multiple code snippets that may have to be inserted in different positions in a code file, or indeed, in many source code files.

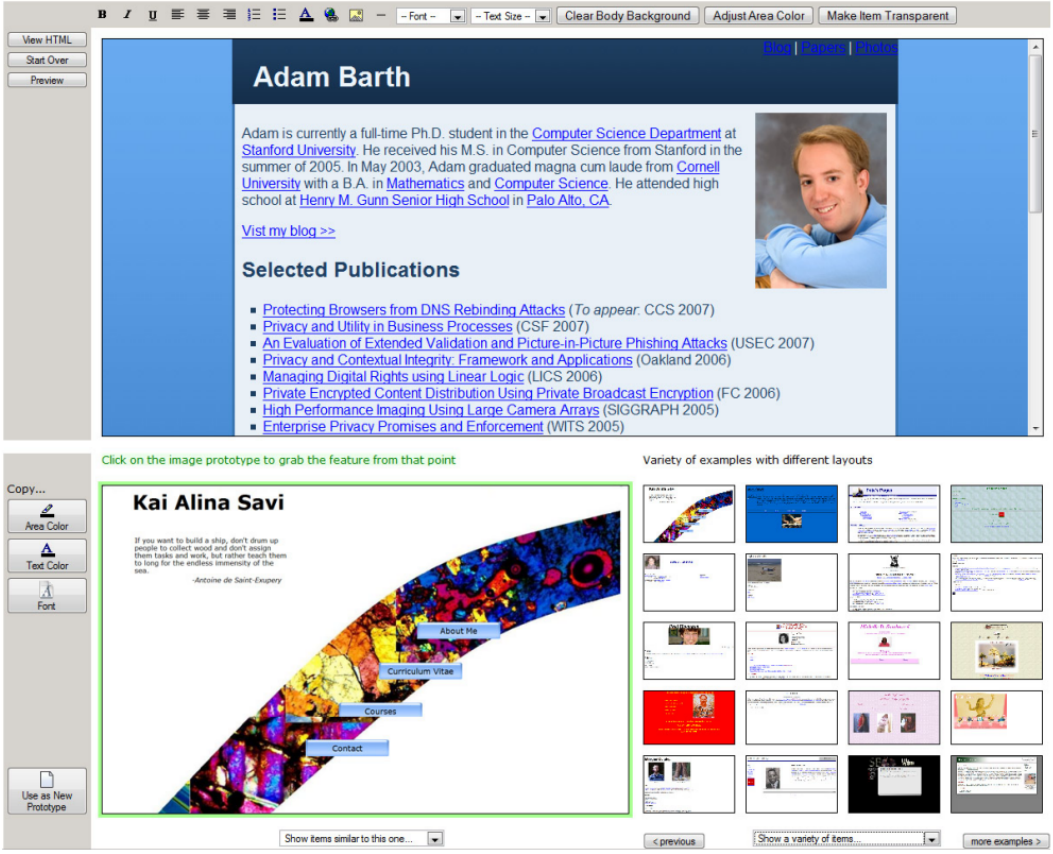
3.2.3 The Adaptive Ideas Design Tool

Examples can provide inspirations during design work.

This research by Lee et al. [2010] focused on the importance of examples for inspiration during web design activities. This owes to the notion that examples show how form and content interact, and thus can raise awareness of potential options of the design space. Adapting existing examples to fit a new context can also help people creatively—and more effficently—handle new situations. However, the way web designers interact with examples currently is mostly informal and ad hoc. Consequently, they mave have problems finding the best examples that could help boost their various design tasks.

Adaptive Ideas let users reuse and modify elements from multiple pages.

Lee et al. then set out to explore the possibiliy of a more structured browsing of examples, and they ultimately came up with the Adaptive Ideas design tool. It is an extension to the Firefox browser's build-in graphical HTML editor, and it is equipped with an interface for parametrically browsing a corpus of example pages. As depicted in Figure 3.12, it consists of three panels: (1) the main, editable area that houses the user's own work; (2) an example gallery that is adaptively-generated in order to have a set of similar items;



Adam Barth

Adam is currently a full-time Ph.D. student in the Computer Science Department at Stanford University. He received his M.S. in Computer Science from Stanford in the summer of 2005. In May 2003, Adam graduated magna cum laude from Cornell University with a B.A. in Mathematics and Computer Science. He attended high school at Henry M. Gunn Senior High School in Palo Alto, CA.

Vist my blog >>

Selected Publications

- Protecting Browsers from DNS Rebinding Attacks (*To appear:* CCS 2007)
- Privacy and Utility in Business Processes (CSF 2007)
- An Evaluation of Extended Validation and Picture-in-Picture Phishing Attacks (USEC 2007)
- Privacy and Contextual Integrity: Framework and Applications (Oakland 2006)
- Managing Digital Rights using Linear Logic (LICS 2006)
- Private Encrypted Content Distribution Using Private Broadcast Encryption (FC 2006)
- High Performance Imaging Using Large Camera Arrays (SIGGRAPH 2005)
- Enterprise Privacy Promises and Enforcement (WITS 2005)

Click on the image prototype to grab the feature from that point

Variety of examples with different layouts

Kai Alina Savi

If you want to build a ship, don't drum up people to collect wood and don't assign them tasks and work, but rather teach them to long for the endless immensity of the sea.
-Antoine de Saint-Exupery

Figure 3.12: The three panels of Adaptive Ideas: the main working area (top), a gallery of samples (bottom right) and a preview for the selected sample. Figure taken from Lee et al. [2010].

(3) a preview pane of the selected example, from which users can borrow design elements (e.g., background color, font, etc.) from. With this tool, users can then reuse and modify elements from multiple pages to create a Web page of their own.

A set of studies were then run to test this approach to designing, and the results suggested that users did prefer to design with the aid of examples. Furthermore, they also found that users also favored an adaptively selected examples rather than random ones, and they often utilized multiple examples when building a new design. Even though this research had a slightly narrower focus to ours, these findings nevertheless influence some of our design deci-

Users prefer to design with the help of examples.

sions, and the idea of using a "gallery" is one that we will explore in some of our design phases, which will be elaborated in a later chapter.

3.2.4 Jigsaw

Jigsaw puts emphasis on small-scale reuse of code examples.

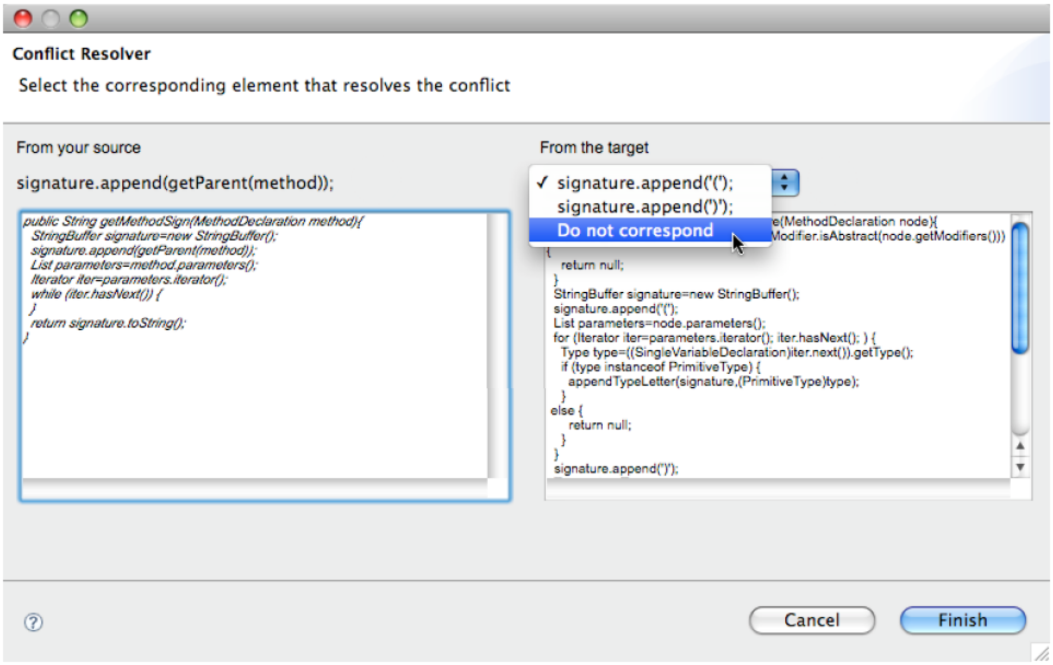
Jigsaw, by Cottrell et al. [2008], is a tool that applies itself to small-scale reuse of code examples. One can easily discern that the most ideal way for integrating code examples is to have all components be flexible and adaptable right from the design phase, but for small-scale uses, the cost in engineering such components often can not be justified. Therefore, one of the biggest hurdles in such small-scale integration process is the contrasting contexts between the original system and the target system. This can range from the more straightforward difference like variable names to the more abstract, high-level context that accompany code, which is often difficult to unveil.

Jigsaw tries to find similarity between terms.

Jigsaw's approach to the integration problems is by finding similarity between the terms (e.g., expressions, statements, declarations) in the original and target systems, with regards to their respective structures and how they are used within their context. Terms with a high similarity value are considered *correspondence candidates*. A conflict will be raised when a term is corresponded with more than one other terms in the other system, and details of this conflict are presented to the users for them to resolve. These correspondences are ultimately used to determine how the terms in the original system should be integrated into the target system (e.g., which terms should be copied, which should be transformed). The tool itself is implemented as an Eclipse plug-in, using the "copy & paste" metaphor, in which the original code is the "copy" seed and the target system is the "paste" seed. Conceptually, Jigsaw allows the developers to focus on the high-level differences between the two systems, while automatically integrating the rest [Cottrell et al., 2008].

Jigsaw lets users focus on high-level differences between original and target systems.

User studies suggested that Jigsaw does have potential in helping developers integrating code examples. When



Conflict Resolver

Select the corresponding element that resolves the conflict

From your source

```
signature.append(getParent(method));
```

```
public String getMethodSign(MethodDeclaration method){
  StringBuffer signature=new StringBuffer();
  signature.append(getParent(method));
  List parameters=method.parameters();
  Iterator iter=parameters.iterator();
  while (iter.hasNext()) {
 }
  return signature.toString();
}
```

From the target

✓ signature.append('(');
signature.append(')');
Do not correspond

```
e(MethodDeclaration node){
Modifier.isAbstract(node.getModifiers()))
{
   return null;
  }
 StringBuffer signature=new StringBuffer();
 signature.append('(');
 List parameters=node.parameters();
 for (Iterator iter=parameters.iterator(); iter.hasNext(); ) {
   Type type=((SingleVariableDeclaration)iter.next()).getType();
   if (type instanceof PrimitiveType) {
     appendTypeLetter(signature,(PrimitiveType)type);
   }
else {
    return null;
   }
  }
 signature.append(')');
```

Figure 3.13: A sample Jigsaw view asking users to resolve a conflict. Figure taken from Cottrell et al. [2008].

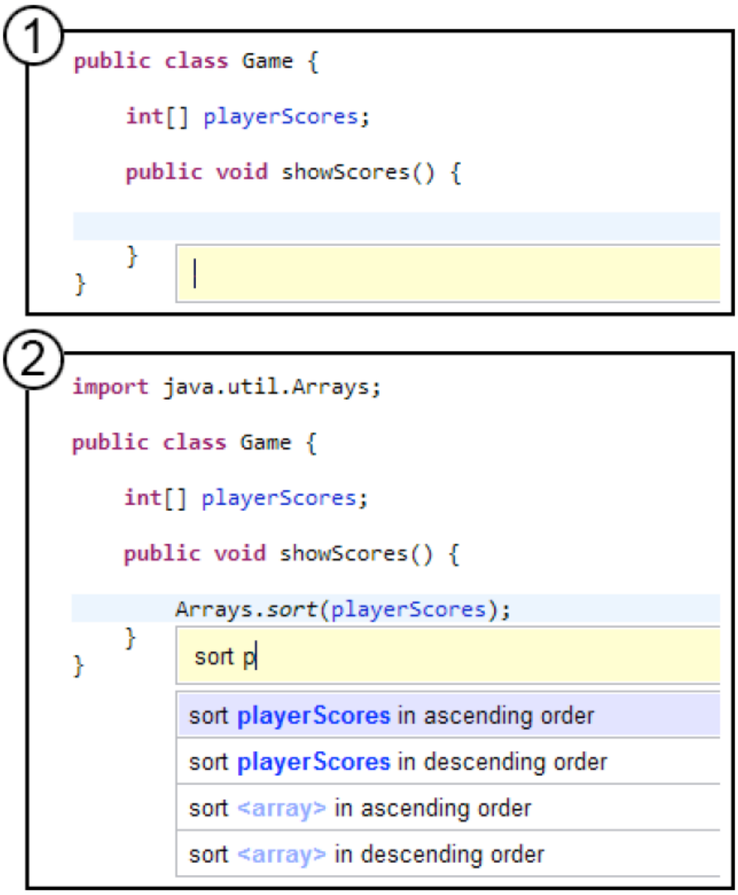
given a certain code example (i.e. the "copy" seed) to incorporate into another code block that has previously been tinkered with (i.e. the "paste" seed), participants were able to reproduce the original paste seed without knowing the result beforehand. However, the studies may have been tailored to favor Jigsaw's strengths. As noted by Holmes et al. [2009], focusing on only structural similarities can often make it cumbersome to configure the original example before the actual integration can take place.

Jigsaw can be difficult to use in real programming environment.

3.2.5 SnipMatch

SnipMatch was introduced to address the various complications that arise during integration of code examples [Wightman et al., 2012]. These problems range from having to combine multiple code snippets, renaming variables, manually locating dependencies, to dealing with the bugs introduced by the code snippets. SnipMatch tries to rectify these problems by targeting two different roles related to

SnipMatch focuses on two roles taking part in code integration: user and provider.



```
public class Game {
    int[] playerScores;
    public void showScores() {

    }
}
```

```
import java.util.Arrays;
public class Game {
    int[] playerScores;
    public void showScores() {
        Arrays.sort(playerScores);
    }
}
```

sort p
sort playerScores in ascending order
sort playerScores in descending order
sort <array> in ascending order
sort <array> in descending order

Figure 3.14: (1) SnipMatch can be accessed directly from the IDE. (2) SnipMatch can tailor search result with user's code context. Shown here is how SnipMatch adds a local variable in the search result. Figure taken from Wightman et al. [2012].

code snippets: users and providers.

SnipMatch uses user's code context to filter and tailor search results.

Code snippet users can access SnipMatch by using a keyboard shortcut directly from the code editing area in their Eclipse IDE, which will bring forth the SnipMatch search box. After users enter the desired search terms, SnipMatch then exploits users' own code context to filter and rank the search results. The code features that are involved in this filtering process are: variable types and names, the cursor position within the abstract syntax tree, program logic, and code dependencies. Results that closely match these features indicate they would require minimal modifications for reuse, and are therefore ranked higher.

Snippet providers use integration markups to define where search pattern parameters should go.

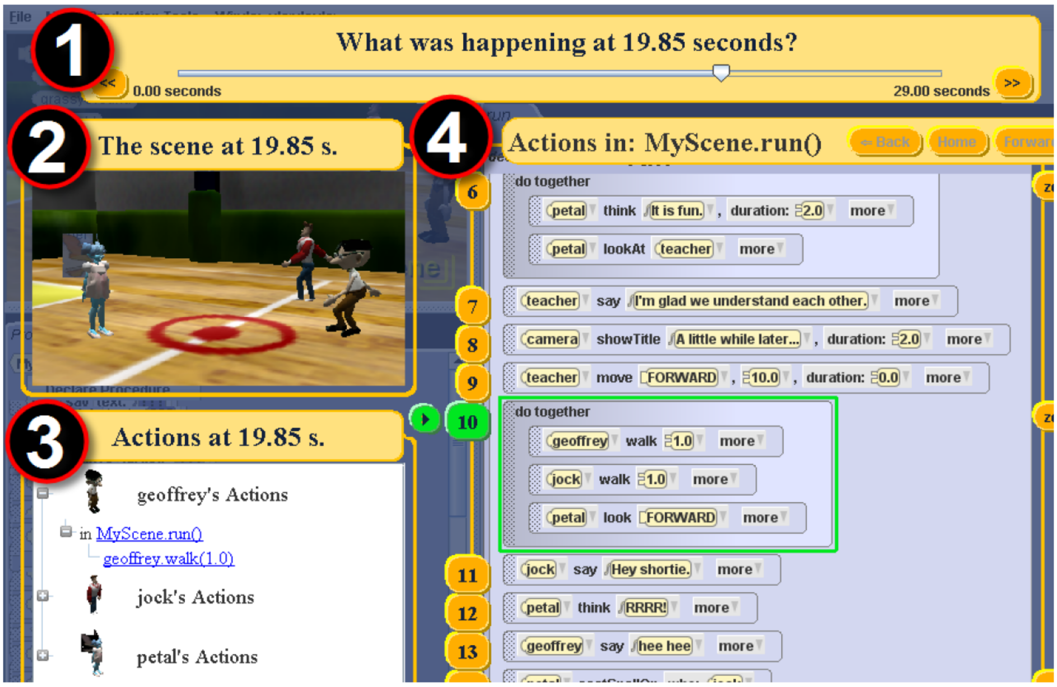
For the snippet providers's benefits, SnipMatch provides an interface for adding and editing code snippets. This interface allow users to modify *search patterns*, and it includes features to facilitate the addition of *integration markups* to the snippet code. Search patterns are the textual description of a snippet, which can contain placeholders for snippet parameters. These parameters will be replaced by snippet user's code context, rendering integration more seamless. Meanwhile, the integration markups can be used to define in which parts of the code snippet the search pattern parameters will appear. They can also be used to import dependencies as required and define preconditions for a snippet to appear in the search results [Wightman et al., 2012].

SnipMatch can reduce time required to search for and reuse code snippets.

SnipMatch has tested positively, with developers found it particularly useful to reduce context switching and as a memory aid. During evaluation, it reduced the time required to settle on and inserting a snippet, and its use of search pattern increased the accuracy of the search results, particularly when compared to the more typical "whole-document" approach to searching. Outside of laboratory-based studies, SnipMatch was also offered publicly and has been used by developers "in the wild," indicating growing interests in tools for integrating code snippets.

3.3 Novel Ways for Programming

This section is dedicated to the tools that offer novel approaches to programming. Indeed, the first two tools were designed for people with no programming knowledge, and thus aim at making coding endeavors as accessible as possible. Still, all three tools also provide support for code comprehension, and the first two do encourage code reuse.



geoffrey's Actions
in MyScene.run()
geoffrey.walk(1.0)
jock's Actions
petal's Actions

```
6  do together
     petal think It is fun. , duration: 2.0 more
     petal lookAt teacher more
7  teacher say I'm glad we understand each other. more
8  camera showTitle A little while later... , duration: 2.0 more
9  teacher move FORWARD , 10.0 , duration: 0.0 more
10 do together
     geoffrey walk 1.0 more
     jock walk 1.0 more
     petal look FORWARD more
11 jock say Hey shortie. more
12 petal think RRRR! more
13 geoffrey say hee hee more
```

Figure 3.15: A sample Looking Glass interface. (1) Time slider for navigating through time. (2) Scene Viewer shows how a scene looks at the selected time. (3) Current Action Pane details what actions characters did at the selected time. (4) Code view selects the executing lines of code. Figure taken from Gross et al. [2010].

3.3.1 Looking Glass

Looking Glass puts focus on story telling instead of code implementation.

Looking Glass is a tool that is targeted at middle school children, to raise their interest in programming [Gross et al., 2010]. Instead of focusing on the technical parts of coding, Looking Glass puts more emphasis on story telling, in that children are encouraged to programatically build the stories they have in mind. Moreover, users can also borrow and reuse others' code without requiring to understand how they work. The intention behind this approach is that by experiencing first-hand what programming can do, children would be willing to spend more time on it, and would eventually learn the technical aspects of coding out of their own desire.

Looking Glass works by having children construct scenarios that will result in 3D animated stories. Users build the scenarios by draging graphical tiles representing programming constructs into the program editor, configuring parameters as needed. Looking Glass is also equipped with a time slider and a preview pane that shows how the scene looks at the selected time, so there is always a heavy focus on the output, as opposed to the code itself. These scenarios are ultimately abstracted as *ActionScripts*, each of which is aimed at a specific *role*. Users who want to reuse an ActionScript only needs to select a character for the role associated with the ActionScript. If they only want to reuse parts of it, they simple observe the output and identify the beginning and ending points of the functionality that they want to reuse [Gross et al., 2010].

Users drags programming components to build *ActionScripts*.

Code reuse is a matter of selecting a character to an *ActionScript*.

During evaluation, all but one students sucessfully captured and reused code. Moreover, the majority of the resulting ActionScripts consisted of more than five lines of code (some even exceeded 70 lines), indicating non-trivial functionalities [Gross et al., 2010]. This result suggests that for code comprehension and reuse, putting the spotlight on the high-level "story" of code, as opposed to its technical implementation details, may be a more natural approach. Incidentally, this idea would actually play an important role during our design process.

Using story may be beneficial for code comprehension and reuse.

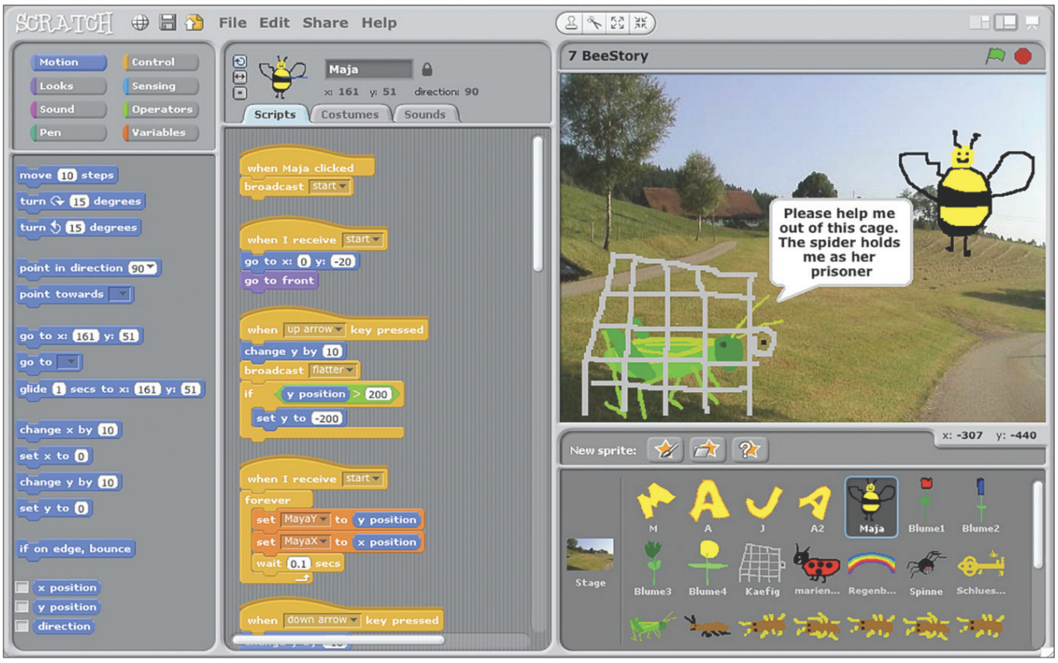
3.3.2 Scratch

The intention behind Scratch [Resnick et al., 2009] was to develop a programming approach that would appeal to people who had never imagined themselves as programmers. To achieve this goal, it has to have a low learning curve, and offers a fun programming experience. Thus, Scratch follows three core principles: make it more tinkerable, more meaningful, and more social than other programming environments.

Scratch's principles: more tinkerable, more meaningful, more social.

Scratch was inpired by kids playing with lego bricks. Kids intuitively snap lego bricks together to build something—an approach that Scratch takes advantage of. Scratch grammar contains a collection of colorful, graphical "programming blocks" that users can use to build programs. The

A scratch program is made of snapping together "programming blocks."



```
when Maja clicked
broadcast start

when I receive start
go to x: 0 y: -20
go to front

when up arrow key pressed
change y by 10
broadcast flatter
if y position > 200
    set y to -200

when I receive start
forever
    set MayaY to y position
    set MayaX to x position
    wait 0.1 secs

when down arrow key pressed
```

Figure 3.16: A sample Scratch interface. The code blocks are designed in a way so that they can only snap together when they make syntatical sense. Figure taken from Resnick et al. [2009].

shapes of the blocks are tailored so that they can only fit when they make syntatical sense. For instance, control structures such as *conditional if* or *loop* are C-shaped to suggest that more blocks should be put "inside" them. Similar to Looking Glass, users of Scratch also

User can reuse other's project in Scratch's online community.

One of the most interesting aspects of Scratch is its vibrant online community, where users can search for and reuse the projects that others have submitted. When someone "remixes" a project, the site automatically adds a link back to the original project, so the original author gets credit. Perhaps it is due to the uses of colorful blocks, but the dominant demography of this site are children of ages between 8 to 16, although there are a good number of adult participants as well [Resnick et al., 2009].

Tools like Scratch indicates that there are benefits that can be reaped by going away from the monotone, textual-based programming. After all, if this technique works for kids with no programming experience whatsoever, then it should also pose little problem to professional developers. Finally, we are also intrigued by the idea of having a stable online community where users can freely reuse other members' work, a feature that we would actually incorporate in our design later.

Non-textual programming may improve code understanding.

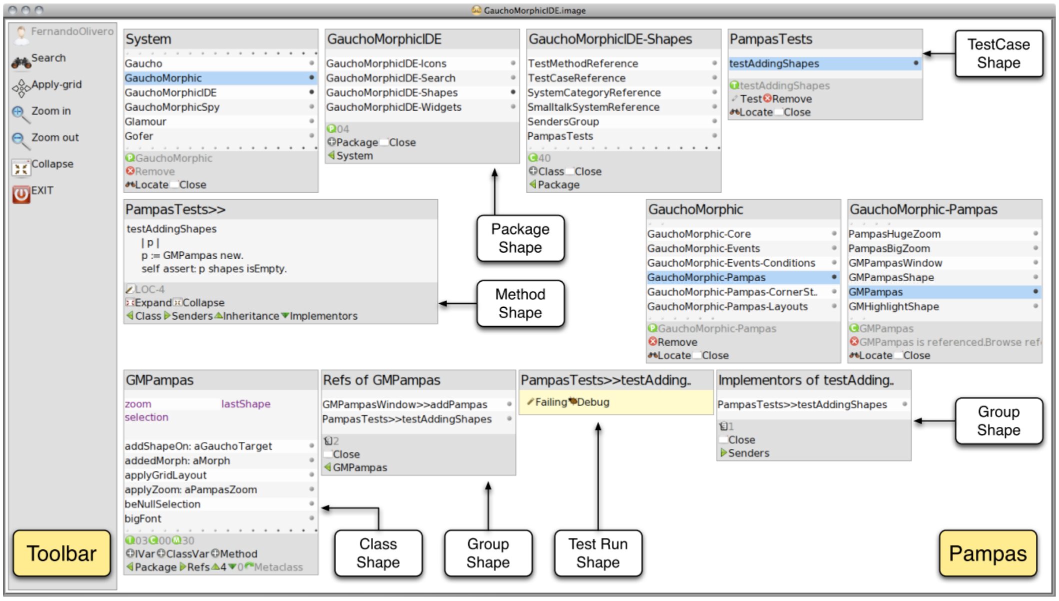
3.3.3 Gaucho

Gaucho aims at steering programming back to modeling.

An overwhelming majority of popular IDEs treat programs as files or texts, and they offer a bunch of tools (e.g., various debugging methods, advanced text editing) to help developers process the code they are working on. This textual approach to programming encourages developers to *write* programs, despite that when talking about creating software, most would use terms like *construct* or *build*. Olivero et al. [2011] argued that these textual representations of software might hinder program comprehension, since developers still have to decode the texts to get meaningful information of how the program actually works. Therefore, they set out to develop Gaucho, an IDE that would steer programming back to *modeling*, and away from writing.

Gaucho is a direct manipulation environment, so users interact directly with objects.

Compared to the textual-based, view-focused approach of conventional IDEs, Gaucho is object-focused Olivero et al. [2010]. It comes with a user interface that borrows generously from the desktop metaphor: one surface area (called the *pampas*), on top of which a group of objects (called the *shapes*) are freely positioned. The shapes are visual representations of various programming components such as packages, classes, methods and developers. Furthermore, Gaucho is a dynamic environment that allows users to move shapes around the pampas, and each shape can be expanded, edited, or deleted as needed. Olivero et al. [2011] posited that Gaucho can ease program comprehension because programmers deal directly with graphical elements depicting software artifacts at high-level views, relieving the cognitive load of having to manually decode written code into said views. Moreover, the relative freedom of the layout makes it easier for developers to make comparisons and understand relationships between components.



```
testAddingShapes
    | p |
    p := GMPampas new.
    self assert: p shapes isEmpty.
```

Figure 3.17: Gaucho User Interface, consisting of a *pampas* and a bunch of *shapes* representing various programming components. Figure taken from Olivero et al. [2010].

Gaucho improves correctness of comprehension tasks, but is slower.

Initial evaluation on Gaucho indicated that it managed to improve the correctness of program comprehension tasks, as compared to conventional IDEs. Each participant was given 14 comprehension tasks that were graded automatically, and of the possible maximum score of 14, users with Gaucho scored an average of 10.4, as opposed to 8.5 of the control group [Olivero et al., 2011]. However, on average participants also took more time to finish the tasks. Usability issues were identified as a primary factor for slowness, particularly the lack of automatic layout or non-overlapping, causing developers spending time to move objects around. Ultimately, tools like Gaucho indicate that a more graphical approach can indeed be beneficial to assist code comprehension.

3.4 Summary

Most of the tools listed in this chapter have a rather specific focus, which means that developers would have to utilize a big number of tools to accomodate various types of tasks. For instance, if they are interested in examining the call sequences among code segments, they may turn to Code Bubbles or Relo, but when they want to acquire a high-level approximation of the overall software structure, Software Cartography offers a more appropriate technique. Since it has already been established that developers have the tendency to neglect these help tools [Roehm et al., 2012], providing them a wide array of choices would ultimately not offer any benefits. Consequently, our aim is to come up with a design for a tool that can address developers' various problems in a more comprehensive manner.

Developers have to use different types of tools to address different types of tasks.

However, these set of tools do propose some interesting ideas and techniques that may benefit our eventual design. Code Canvas, for example, with its zoomable working area, offers a less restrictive method of navigating a code project, while Code Thumbnail provides an elegant, almost simplistic fix against *visual uniformity*. Tools like Blueprint and Codelet also highlights the importance of contextual information that accompanies code examples, which allows users to quickly decide on the most relevant information. And finally, novel systems like Looking Glass and Scratch bring along a fresh perspective towards programming that pulls the focus away from technical details, which is something that we would explore more during our design process. The next chapter will ease us into the design period by first establishing the scope of our research as well as the characteristics that our eventual design would have to exhibit.

Some interesting techniques and ideas from this chapter are going to be incorporated into our own design.

Chapter 4

The Research Base

This thesis work tries to bridge the gap between software comprehension and information foraging.

Based on the way they are categorized, it is clear that most of the tools presented in the previous chapter only provide developers with some assistance in either *code comprehension* or *information foraging*. Indeed, the only two tools that support both stages are Scratch [Resnick et al., 2009] and Looking Glass [Gross et al., 2010], both of which are intended for programming newbies and are therefore quite disconnected from conventional coding activities. Thus, it can be inferred that there is an existing gap between the two groups of programming activities. The main goal of this thesis work is bridging this very gap, which is done by proposing a novel *design* for a tool that is able to address developers' problems more comprehensively.

Instead of focusing on the developers in need, we are more interested in the creation of information that is going to be consumed by said developers.

We approach this conundrum through a somewhat contradictory route. It is true that the ultimate goal of this research is to help the developers who are having problems comprehending code or gathering relevant information. However, at this initial stage they are not given the main focus. Instead, the spotlight is tilted in the direction of the developers who compose and ultimately share the information that is consumed by the former group of developers. Naturally, these two groups of developers are not mutually exclusive, as most software developers would find themselves alternating between the two groups at one time or another. For the sake of convention, from here on out we would dub the former group as *consumers*, and the latter as *composers*. Furthermore, we would refer to the pieces of information offered by composers and used by consumers as *compositions*.

Definition: *Consumers*

CONSUMERS:
Developers who search for—and subsequently reuse—some pieces of information.

Definition: *Composers*

COMPOSERS:
Developers who create and publish pieces of information that are used by consumers.

Definition: *Compositions*

COMPOSITIONS:
Pieces of information offered by composers and used by consumers.

A composition can serve as the solution to a code comprehension or information foraging problem.

Our reasoning for going this route is the notion that solving developers' problems, be it in the realm of code comprehension or code reuse, would essentially require the exact same procedure: the creation of a solution, or in our case, a composition. We have mentioned in one of the previous chapters (specifically chapter 2 "The Lives Of Developers") that when trying to understand code, developers often build a mental model of the code to help them grasp the implicit knowledge that is inherently contained by the code [LaToza et al., 2006]. This statement can be rephrased by stating that a mental model is a solution that is specifically *composed* to solve the problem of comprehending code. Similarly, when developers perform information foraging, the pieces of information that they ultimately choose to sample (e.g. tutorials, code snippets) must have been *composed* by someone else. We can then say that in our design scheme, a composition is a direct answer to a developer's problem, regardless of its type. This would indicate that the root of a composition is a problem or a task.

Incidentally, our approach also falls in line with the finding that at any given time, most developers are typically occupied with a certain task, prompting them to focus only on the parts of a code project that is relevant to said task, instead of trying to understand the entirety of the project

[Singer et al., 1997, Roehm et al., 2012]. It is a safe bet that a task that is dominating a developer's attention would contain some problems that need solving, and this brings us back to our approach that revolves around problem-oriented compositions. Hence, it is our belief that putting the focus on the creation, and subsequently the distribution, of compositions is a proper way to kick off our research. Our design goal is to propose an efficient method for creating and sharing a composition, regardless of the type of programming activities involved.

Our goal is to propose an efficient method for creating a composition, regardless of the type of programmming activities involved.

Before elaborating on our actual design process, we will first establish a set of design principles that would serve as our foundation throughout the rest of this thesis work. These are then followed by a set of secondary properties that further mold the shape of our design.

4.1 Design Principles

In order to better orient ourselves during the tumultuous design period, a set of design principles are established. These principles outline a set of features and characteristics that the proposed tool design should exhibit to become the panacea for developers' woes. Ultimately, a tool that can assist developers in both code comprehension and information foraging should:

Three design principles are introduced to help guide our eventual design.

- Emphasize the visual aspect of coding
- Support lightweight creation and reuse of compositions
- Provide compositions with their task-related context

The following three sections will further elaborate on these criteria.

Visual Emphasis

Visual representations can help reduce code uniformity.

Taking a quick scan at the available tools aiming to help code comprehension, the importance of this principle becomes apparent. An overwhelming majority of these tools come up with visual representations of software systems in all sorts of shapes and forms. For instance, from the tools presented in this paper alone, software systems have been portrayed as cartographic images, "bubbles" containing small code segments, thumbnail images of code files, and so forth. It can be deduced that this is a natural respond against the epidemic of *code uniformity* highlighted by DeLine et al. [2006], which is an inevitable repercussion from relying too heavily on textual representations during the "writing" of code.

Visual representations can provide better outlooks of abstract concepts than textual ones.

Moreover, as pointed out by Olivero et al. [2011], developers often decode written code back into its higher-level, more abstract concept in order to gain a better grasp of said code. Although this abstract information can be described using text, users are typically required to read through the whole descriptive passages before comprehension can occur. A better alternative would be to provide a "snap shot" of a concept, which can also mirror the mental model that developers instinctively construct while inspecting code, and this can be better captured using graphical entities such as graphs or diagrams. Accordingly, based on these two reasons, we believe that a composition should favor graphical over textual representations of knowledge.

Portability

Our design should exhibit clear and straightforward features that are easy to use.

One of the major motivations behind this principle is the notion proposed by Roehm et al. [2012] that in the occasions that developers make use of software comprehension tools, they rarely do so in the most optimal manner, lacking the knowledge about even the standard features of these tools. Consequently, we strive to model our design into one that exhibits clear, straightforward features that are easily understandable and accessible. Since a big part of our design scheme involves the creation of compositions, an ease of creation is therefore crucial, and the inclusion of this principle becomes a necessity.

Simple creation and reuse of compositions are both important elements of our design.

Bruch et al. [2010] also provides some inspiration with their research on IDE 2.0, which he intended to be an improvement over IDE 1.0, not unlike how Web 2.0 is considered a more advanced version of Web 1.0. Just like its Web counterpart, IDE 2.0 encourages users to interact and share knowledge with one another. The driving force behind IDE 2.0 is to take advantage of a collective knowledge base, and one of its main principles is to encourage developers to build their services on top of existing services by providing easy-to-use APIs. Users are then encouraged to avoid reinventing the wheel and simply reuse existing information (in this case, existing APIs), an aspiration which is very similar to our intention of having consumers reuse compositions. This convinces us that on top of a lightweight creation method, a simple method for reusing compositions plays a similarly significant role in our design.

Task Oriented

Code context is an important aspect of both code comprehension and information foraging.

Code context is an essential aspect of code comprehension, since it provides the high-level concept that dictates the way code is written. Simply put, developers cannot possibly acquire total understanding of code without looking into its context. Additionally, code context is also a necessity during information foraging. Starke et al. [2009] discovered in their study that when performing searches, developers typically only explore a small number of matches. Indeed, the most commonly observed behavior was for developers to investigate exactly one result, or none at all. Based on this finding, it is then clear that our design should equip a composition with a clear context, and to further include this context in the search results. The importance of context has also been realized by tools such as Snipmatch [Wightman et al., 2012], Codelet [Oney and Brandt, 2012], and Blueprint [Brandt et al., 2010].

However, we would like to push this concept even further. Taking an inspiration from the tool Looking Glass [Gross

Our design puts more emphasis on a composition's task-related context than its code-level details.

et al., 2010], a composition in our design scheme puts more emphasis on its high-level, task-related context as opposed to the microscopic, code-level details. The motivation for this approach again originates from the finding that we have cited multiple times now: developers routinely approach code based on their tasks, meaning they are only interested in the parts of code relevant to their current tasks. [Singer et al., 1997, Roehm et al., 2012]. Our hypothesis is that since every task is inevitably driven by a high-level purpose, it is a more natural approach to revolve the process of creation, distribution, and reusing of a composition around its high-level, task-centered information.

4.2 Secondary Properties

The three principles described in the previous section make up the principal traits that our design should possess. Here we list some additional attributes that would further guarantee the effectiveness of our proposed design.

Accessibility

Accessibility can be achieved by taking advantage of the concepts that users are already familiar with.

A tool with incredible features would not amount to much if its users have no idea how to access said features. Some tools demand a lot of time and cognitive investment on the users' parts just to learn all the basic features, which may eventually discourage a large portion of users from using them. In other words, ease-of-use is key, and one way to achieve it is by taking advantage of the various paradigms that users are already familiar with [Girgensohn and Shipman, 1992]. This includes, for instance, providing basic shapes such as arrows and boxes to build a composition, since they already have a prominent presence in the various sketches produced by developers during software development [Cherubini et al., 2007]. This property ensures that our design would help developers solve their problems with as little distraction—and effort—as possible.

Modifiability

This property serves as a support system to one of the main design principles that calls for lightweight reuse of compositions. It turns out that when developers reuse a certain piece of information (e.g. in the form of code snippets), it is a highly common occurence that they alter the information instead of reusing it right away, especially when there is a divergence in context between the original source and their own code [Cottrell et al., 2008, Wightman et al., 2012]. Similar cases may occur in our proposed scheme when developers locate a composition that closely, but not exactly, addresses their problems. Developers may prefer to alter said composition rather than to keep looking for the one that matches their exact needs, and hence, it is crucial to allow developers to perform modification on existing compositions.

Developers often alter an existing piece of information before they integrate it into their own work, making it an important factor in our design.

Collaboration-Friendly

According to Cherubini et al. [2007], developers regularly make use of diagrams in their code comprehension pursuit, a tendency that is already supported by our first design principle. However, diagrams are also often employed to *design*, for instance to create a new subsystem or to manipulate existing structures. The design process often involves collaborative brainstorming efforts from multiple persons, in which they go back and forth on ideas that gradually build up the design. For this reason, our proposed design should support collaborations during the creation and manipulation of compositions.

Developers often work together to build a diagram, particularly during a design phase, so our design needs to support this tendency.

Spatial Freedom

This property stems from the observation that developers tend to regard diagrams that are automatically-generated by tools as less “interesting” than the ones that are created manually through collaborative efforts [Cherubini et al., 2007]. Indeed, this inclination may be one of the reasons

Developers are more interested in the diagrams that are not automatically-generated.

that developers often forgo software visualization tools during code comprehension tasks, an observation raised by Roehm et al. [2012]. At any rate, these findings suggest a tool that avoids automatic placements of objects and in its place, encourages spatial freedom while constructing the content of a composition.

Recordability

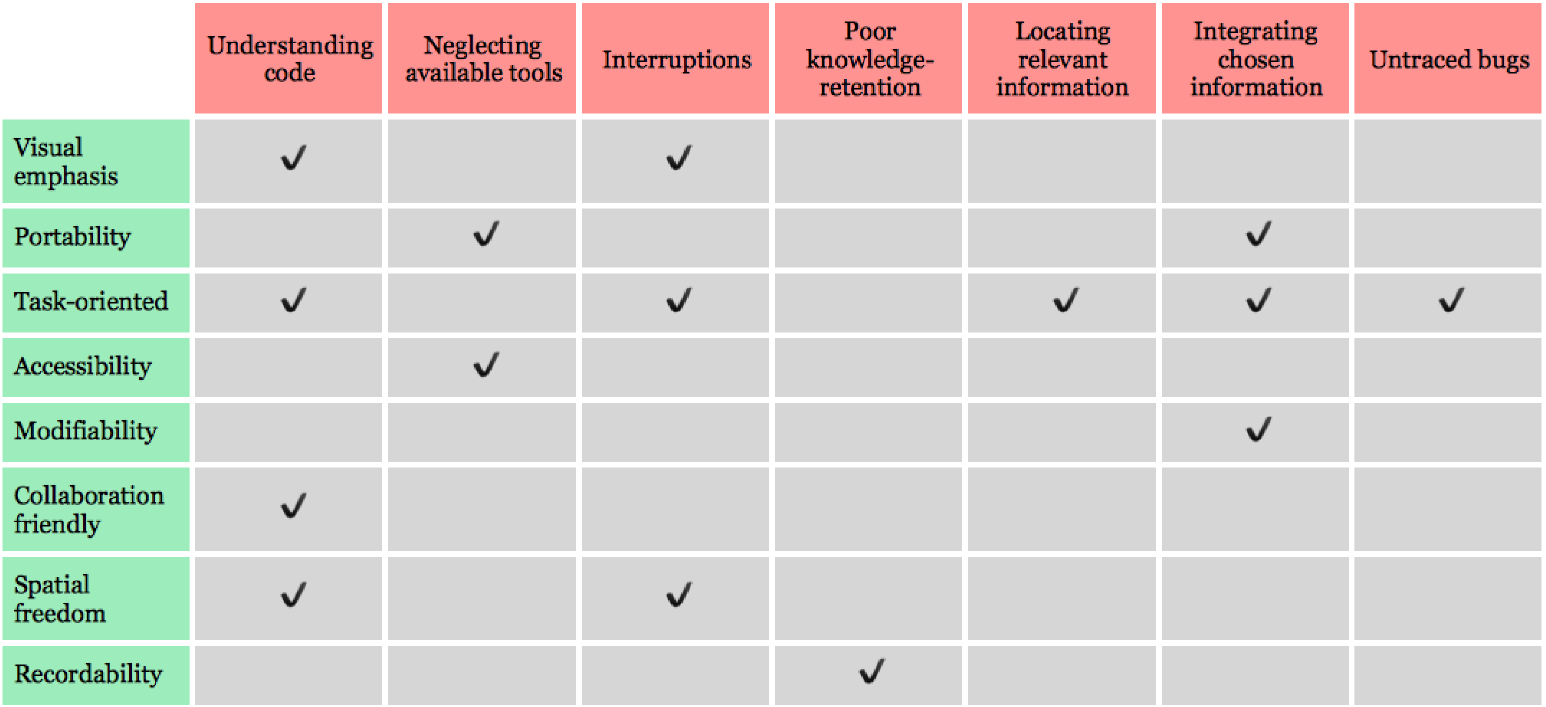
Our design should let users have multiple versions of a composition.

Many studies have noted that developers rarely make an official document of the knowledge they acquire during code comprehension [LaToza et al., 2006, Roehm et al., 2012], which diminishes the durability of said knowledge. Instead, developers utilize personal notes or sketches to help them externalize abstract ideas, but these are transient in nature due to the cost of transforming these physical objects into electronic versions. Since our proposed design already revolves around computer-generated compositions, making an official record of a composition is a natural feature. To improve productivity, this feature should also be extended to allow for multiple recordings of a single composition. This may be beneficial for when developers prefer to have multiple versions of a composition, so they have the ability to track the evolution of a composition until it reaches the final version.

4.3 Summary

Our proposed principles and properties address the problems that developers face.

Figure 4.1 illustrates how our proposed design principles and properties deal with all the problems that have been established in chapter 2. We directly address the major problem of "understanding code" by providing a *visual* outlook of the code and highlighting the *task-related* information, which would help facilitate developers' mental model creations of said code. Letting them *collaborate* in *freely constructing* said mental model would further ease them into understanding unfamiliar code. Also, since the problem of "interruptions" typically is a by-product of the former problem, it is also inadvertendly addressed by most of the



	Understanding code	Neglecting available tools	Interruptions	Poor knowledge-retention	Locating relevant information	Integrating chosen information	Untraced bugs
Visual emphasis	✓		✓				
Portability		✓				✓	
Task-oriented	✓		✓		✓	✓	✓
Accessibility		✓					
Modifiability						✓	
Collaboration friendly	✓						
Spatial freedom	✓		✓				
Recordability				✓			

Figure 4.1: How our design principles and properties address each of developers' problems during a maintenance task.

same principles and properties, save for the one that encourages developers to work together with (i.e. interrupt) one another.

Further details on how our tool would help developers.

In the meantime, by enforcing *portability* and *accessibility* in the design, we ensure that developers would not ignore our tools when they stumble into problems. Similarly, the property of *recordability* would help them retain valuable implicit knowledge that they have gathered during a code comprehension period. Furthermore, *task-oriented* information, on top of improving code comprehension, can also help developers in selecting the most relevant information, since it would help reduce ambiguity among technically similar compositions. Proper understanding of the context of a composition would further help developers in easily integrating its content into their own work and subsequently minimize unexpected bugs.

If our design can fulfill these principles and properties, then we can achieve of design goal.

Ultimately, this section has highlighted the potency of our selected design principles and properties. This implies that if our design can fulfill these principles and properties, then it would be able to help get rid of the various stumbling blocks that developers might encounter during maintenance tasks. In the next chapter, keeping these principles and properties in mind, we would start the actual process of designing a system that would assist developers in their code comprehension and information foraging pursuits.

Chapter 5

The Design

The previous chapter saw us laying out the groundwork for our research, from the general purpose and approach of our work, to some more detailed principles and properties that are essential for the eventual success of our design. Based on this foundation, the main part of thesis work has been reached: the actual design of an application that can help developers overcome various problems during software development, particularly during maintenance tasks. We shall call this application *Code Mixer*, with the idea that our design should eventually allow users to easily adjust and integrate (i.e. mix) others' code with their own.

The application, dubbed *Code Mixer*, should allow users to easily mix others' code with their own.

This chapter consists of four sub-chapters, each corresponds to one iteration of a design phase. Each phase is structured similarly: it opens with some general information and the thought process behind it, which then segues into a detailed description of the interface, followed by some illustrations on how each user role (composers and consumers) would interact with said interface, and finally it caps off with some assessments that also set the ground for the next design phase. These phases build incrementally over time and the final version of Code Mixer would ultimately combine various elements from the previous design phases.

The design period consists of four phases which grow incrementally.

5.1 Incorporating Design Patterns

This design phase is mainly concerned with the third design principle and the information foraging activities.

The launch of our design is centered around the third design principle, namely *task orientation* (see Section 4.1 in p. 49), and the *information foraging* activities; we would eventually address *code comprehension* in the later design phases. Our first inspiration arises from the observation that most information foraging help tools do not supply code examples with sufficient contextual information, and this may potentially hinder developers' decision-making process and long-term knowledge retainment. Indeed, even when a code chunk is equipped with context, as presented by Codelets [Oney and Brandt, 2012] and Blueprint [Brandt et al., 2010], this information is largely stand-alone, isolated from anything outside the scope of the code chunk itself. Consequently, most existing tools only supports integrating a single code chunk at a time, and lack the support for solutions that span multiple code chunks.

Our approach is to let developers focus on the tasks instead of code details, which seems to work for programming newbies.

While it is certainly beneficial to understand *what* a code chunk does, we also think that understanding *why* it exists in the first place, particularly in relation to the *task(s)* that it pertains, is equally important. From the list of tools in Chapter 3 ("Related Work"), only two seem to share this idea: Looking Glass [Gross et al., 2010] and Scratch [Resnick et al., 2009]. As noted previously, both tools are targeted at those who have no programming experience whatsoever, and instead of tutoring the users on the nitty-gritty of programming protocols, they let the users focus on the end results of the code (i.e. the tasks). Both tools have proven to be quite effective, and Scratch especially boasts a thriving online community in which more than 3 million projects have been shared by its members. As a side note, an argument can also be made that The Adaptive Ideas [Lee et al., 2010] also puts the end result front and center, but this tool's scope is limited to web design, whose main elements of HTML and CSS call for less structural complexity than what is typically required by a lot of software solutions. Ultimately our hypothesis is simple: if the task-focused programming approach works for beginners, it should work for experienced programmers as well.

It is crucial to come up with a standardized convention to structure task-centered information.

The very first concern arising from our selected approach is to come up with a standardized structure or format to ground task-centered information. We argue that the main reason code-centered approach is popular among information foraging help tools is because it offers a straightforward method to reuse information, which can be achieved by as simple as copying a code chunk into the desired location in the users' own code. A task-oriented approach like ours, however, is a tad more problematic, mainly due to the fact that task descriptions are largely free-form: there are no definite rules as to how to define them. Tasks can be defined in the forms of graphs, bulleted lists, narratives, a mixture of multiple formats, or indeed, they can even be written in different languages. We believe that a certain structural convention that can properly delineate tasks descriptions is crucial in order to facilitate the process of searching, selecting, and most important of all, reusing a composition.

In order for knowledge transfer to suceed, the problem space should be clearly modeled, which can be achieved by using design patterns.

A similar research by Lung et al. [2002] has once explored the idea of knowledge transfer by way of analogy. Analogical reasoning, which is a technique that deals with finding similarities between multiple entities, is often employed during the mapping of a solution from a well-known problem to a new one. This is a paradigm that closely resembles our design scheme, in that when a consumer borrows a composition, knowledge transfer then occurs between the composer of said composition and the consumer. This research also agrees with our hypothesis that in order to achieve effective knowledge transfers across multiple applications or domains, the problem space (i.e. the tasks) should be explicitly and clearly modeled, as opposed to putting all the focus on the solution space (i.e. the code chunks) [Lung et al., 2002]. Ultimately, the most interesting part of this research is the proposition that *representation* holds a major role in identifying analogous problems and solutions, and that *design patterns* would do a fine job playing this role.

Design patterns themselves are aimed at capturing recurring problems and their solutions [Borchers, 2000], and though initially devised as an architectural technique, they have since been adopted by other fields, including software

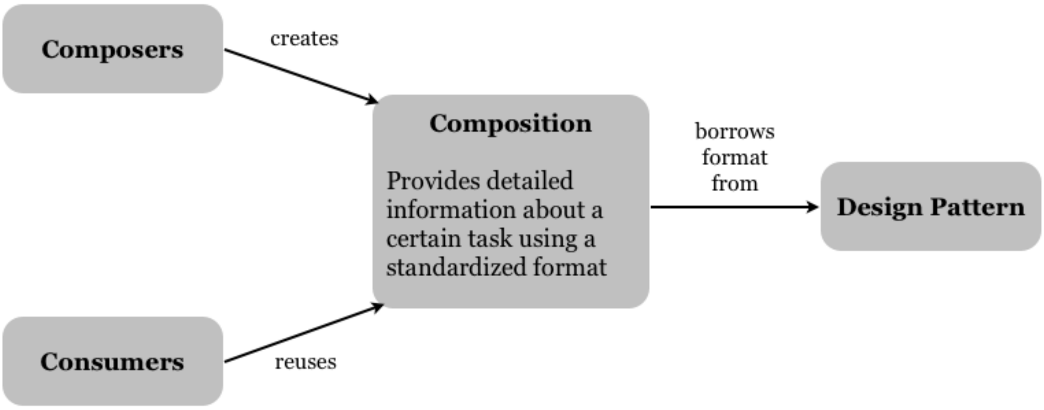

Composers
creates
Composition
Provides detailed information about a certain task using a standardized format
borrows format from
Design Pattern
Consumers
reuses

Figure 5.1: Our first design phase centers around the composition, which borrows the format of design patterns.

Design patterns aims at capturing recurring problems and their solutions, and their benefits in software engineering have been shown by several studies.

engineering. For instance, Baggetun et al. [2004] once investigated the possibility of using design patterns to assist knowledge transfer in collaborative learning. Collaborative learning may benefit from a uniform method for communicating ideas because the creators and owners of e-learning knowledge typically originate from multiple backgrounds, such as computer science, psychology, social science, and so forth. Ultimately, it was concluded that patterns can indeed mediate knowledge and transfer knowledge even among participants with different levels of expertise [Baggetun et al., 2004]. Furthermore, De Rore et al. [2009] also conducted an experiment in which they asked a group of developers to finish a few UML modeling tasks. One group of participants was exposed to a set of Software Design Patterns during the study while the other group was not. It turned out that exposure to patterns did yield a significant improvement to the resulting models. Moreover, it was observed that after participants took some time to learn and apply the patterns, they eventually managed to correct and improve their models.

Code Mixer would only borrow the design pattern formats instead of trying to emulate its true essence.

The findings from the aforementioned studies have convinced us to incorporate design patterns in our design scheme, specifically to structure task-related information and facilitate the distribution of compositions. Nevertheless, our main focus does not lie in emulating the true essence of the technique as intended by its creator Alexander Christopher, which is to achieve what he dubbed "the quality without a name," which is an ideal quality to an environment that one should strive for [Alexander, 1979]. Instead, we are only borrowing the design patterns *format* to be used as a foundation to a composition. It can then be said that a composition has a narrower scope than a typical design pattern, since a composition's main purpose is merely to provide a standard structure for task-related information, so that it can be easily shared among software developers.

Code Mixer differs from Software Design Patterns in that it tries to be as accessible to everyone as possible, even beginners.

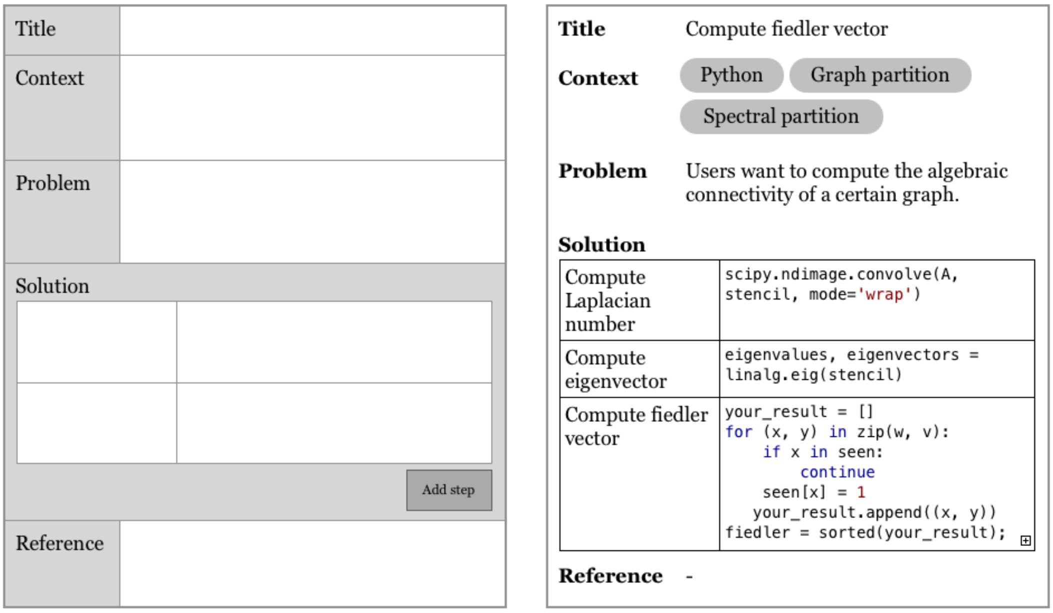
Moreover, it should also be noted that compositions in Code Mixer are also different from *Software Design Patterns* as conceived by the Gang of Four. Software Design Patterns are typically quite intricate and often require a certain level of knowledge and expertise before they can be properly applied. Code Mixer, on the other hand, is intended to be accessible to developers with all skill levels. Incidentally, this is more in line with the way Alexander devised his patterns to be understandable to even non-architects, with the purpose of letting the inhabitants (users) themselves collectively build the environment they live in [Alexander et al., 1977, Borchers, 2000].

The Form

In this phase, Code Mixer is a web-based environment and a composition is akin to a single design pattern.

This initial design phase establishes two major characteristics of Code Mixer. Firstly, it acts as a *web-based environment* that hosts and facilitates the creation of compositions. Secondly, a composition is comparable to a single design pattern, and hence boasts the following structure:

- *Title:* serves as an identity of a composition.
- *Context:* provides conditions that should to be fulfilled before a certain composition is executed.
- *Problem:* describes the complications (i.e. the real life tasks) that a composition is trying to address.
- *Solution:* describes the measures that have to be taken to fix the problems illustrated in the previous section.



Title

Context

Problem

Solution

Reference

Title Compute fiedler vector

Context Python Graph partition Spectral partition

Problem Users want to compute the algebraic connectivity of a certain graph.

Solution

Compute Laplacian number

```
scipy.ndimage.convolve(A,
stencil, mode='wrap')
```

Compute eigenvector

```
eigenvalues, eigenvectors =
linalg.eig(stencil)
```

Compute fiedler vector

```
your_result = []
for (x, y) in zip(w, v):
    if x in seen:
        continue
    seen[x] = 1
   your_result.append((x, y))
fiedler = sorted(your_result);
```

Reference -

Figure 5.2: An empty form for building a composition (left) and a sample composition with its filled out content.

- *Reference:* outlines the components (e.g. other compositions, libraries) that can be used to complete the current composition.

Each component from the list has its own fields, save for *solution*, which typically consists of multiple field pairs.

As we have mentioned previously, the design patterns format is merely used to provide a structure to the typically form-less task description. A composition is essentially a task description that is broken down into smaller pieces, with each piece corresponding to one component from the list above. Each component of a composition, aside from the *solution*, has its own "field" that simply contains some information describing its role in the composition. Meanwhile, the *solution* component itself typically consists of multiple field pairs, with each pair represents one "step" of the solution. The first of the field pair contains the general description of the step and the second one is for the actual content. The latter is comparable to the typical code chunks that consumers can copy and further modify as needed, but it may also contain non-code instructions such as configuration information, hardware settings and so on. However, a step may also contain non-code information, such as

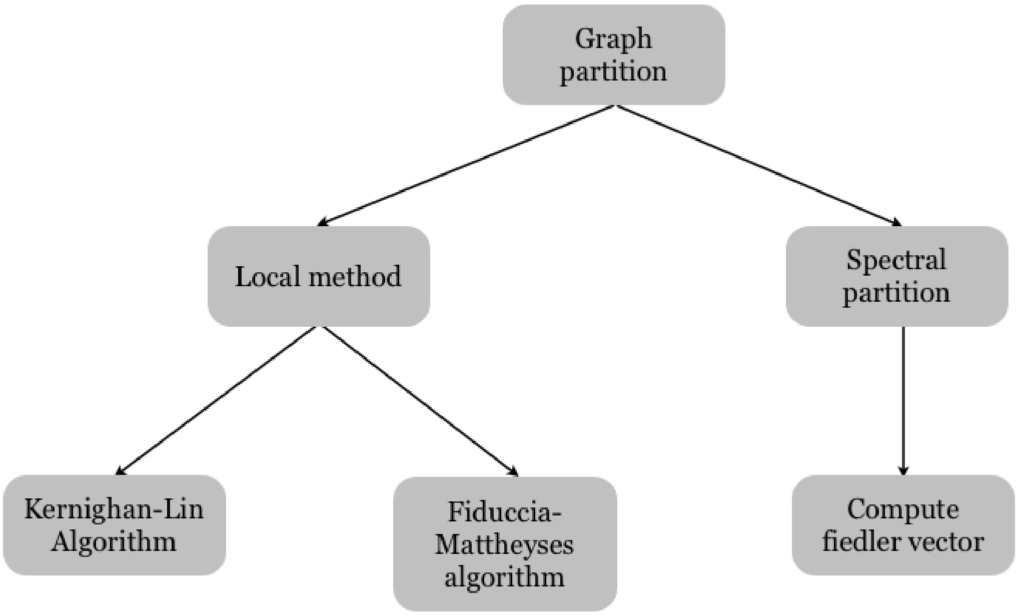


Figure 5.3: A sample information network that revolves around the domain of *graph processing*.

configuration information or even hardware related operations. It bears mentioning that some components, namely the *problem* and the *solution*, may also contain visual objects such as tables and graphics on top of texts, and that all components, save for *reference*, are mandatory. Similar to a conventional design pattern, the purpose of these visual objects is to provide a quick overview on the information contained by a certain component.

Compositions can be connected with one another to form a graph that roughly represents a certain domain.

Another pattern-like characteristic of a composition is how it can be connected to other compositions to form an *information network*, which is basically a network of compositions that deals with various problems within a roughly similar domain. For instance, tasks that deal with mathematical calculations would belong in the same graph, while tasks that are related to graphical operations may belong to another graph. Users can create these connections manually, but Code Mixer can also automatically create the connections by inspecting the context and reference components of the compositions. If a composition is included in the context of another composition, the former composition is placed at a higher level in the graph. Meanwhile, an inclusion in the reference indicates a lower level placement in the graph. The resulting structure of these connections is a graph that may provide a visual representation of a certain task domain. We believe this feature can help users find the most relevant information to their problems, since the basis of this networks is essentially a group of high-level, real world tasks.

The Composer Workflow

The composer workflow is not drastically different from building a conventional tutorial.

To build a new composition, a composer simply needs to visit the Code Mixer web environment and proceeds to fill out the form following the design patterns structure. The content can be typed in manually or copied from another source, to accommodate such cases in which the solution contains code chunks from the composer's own work. Furthermore, composers can also insert graphical objects by uploading them from their own file system or linking to existing objects on the web. Ultimately, this is not drastically different from building a *conventional tutorial* on the Web, the only exception being that a composition requires a more standardized format.

Definition: *Conventional Tutorial*

Conventional Tutorial:
A piece of writing that aims to illuminate on a certain topic and typically contains a step-by-step guide to perform a task. For our purposes, this term specifically refers to such articles published on the Web that talk about programming-related topics, and typically contain code snippets.

The Consumer Workflow

Consumer would reuse a composition the same way they would follow a tutorial on the web.

Similarly, there is little change in the way a consumer would recycle a piece of information from a web source. Referring back to the web tutorial analogy that has been employed in the composer section, a consumer is given full freedom in deciding the steps of a certain composition that should be copied (i.e. not all steps have to be executed), depending on their need. It is also entirely depends on the consumer as to where in the working code a certain code chunk should be copied. The only slight adjustment occurs once a code chunk is pasted into the consumer's code, where a comment would be inserted on top of the reused chunk, stating the source of the information (i.e. the web address of a certain composition), a technique adopted from *Blueprint* [Brandt et al., 2010]. This way, it is relatively easier to trace back a composition to its original source to verify a code chunk's intentions.

Self Assessment

One of the biggest drawbacks of the current version is that it is too demanding, especially for composers.

The resulting design of the current phase merely serves as a kickoff to the whole idea of Code Mixer, and therefore it may seem rather unrefined, crude even. One of the major drawbacks of the current version is probably how it requires a lot of effort from—and put a lot of trust on—the composers. We are basically asking the composers to write a complete, proper pattern every time a composition is built. Wightman et al. [2012] has noted that when a person has decided to invest some time in building a reusable piece of information, he or she would not mind the extra effort it requires to make it as meticulous as possible, and for this type of user, Code Mixer should work just fine. But for the rest of the users who are looking for a lightweight method to share information, this complexity would eventually discourage them from using the system, which means that we have violated one of our design principles regarding *portability* (see Section 4.1 in p. 49).

The current scheme barely improve the typical information foraging activities.

Another glaring problem is the fact that, as we have indicated earlier, the current scheme does little to enhance the typical information foraging activities. Users will still need to go back and forth from their work environment to the web browser, an approach that may dampen their work momentum.

Nevertheless, the current version does propose a more standardized format for task descriptions and the ability to

Providing users with task-oriented information would help them select and integrate relevant information.

create a visual network of high-level information. We believe that with these features, Code Mixer can help users to decide on a relevant piece of information, since the somewhat restrictive format of design patterns would allow little room for ambiguity. Not to mention that once they gain a better understanding of the task-oriented properties of a certain composition, drawing an analogy to their own problems would become a much simpler task and ultimately, integrating and tailoring the code chunks contained by a composition would also become a less taxing endeavour.

5.2 The Pattern Language

Code Mixer is no longer a fully web-based environment, but instead consists of a desktop application and a web-based repository.

One of the bigger problems carried out from the previous design is the lack of improvement of users experience regarding their interaction with a Web-based content. Consequently, one of our immediate concerns in the current design phase is to devise a more seamless interaction method, one that would severely reduce the need to return to the web browser over and over again. Our first big decision, accordingly, is to discard the idea of a fully web-based environment and in its place, we propose a desktop-based application for Code Mixer that should be accessible from inside the IDE. Nevertheless, the Web itself is such an integral and convenient platform, so its role is not completely eliminated. On top of the desktop application, we employ the Web to serve as *central repository* that would host the compositions, an idea inspired by Scratch [Resnick et al., 2009], and to a lesser degree, The Adaptive Ideas design tool [Lee et al., 2010]. The idea is that users would interact with the desktop application from inside the IDE, and the desktop application would interact with the repository in the background, acting as the bridge between the users and the compositions. The central repository would further act as a collective knowledge base and also as a “gallery” of compositions that users can browse through, not unlike the currently popular concept of *App Store.*

Another crucial tweak in our design involves an adjustment of the scope of a composition (i.e. a composition represents a single design pattern), which is heavily inspired

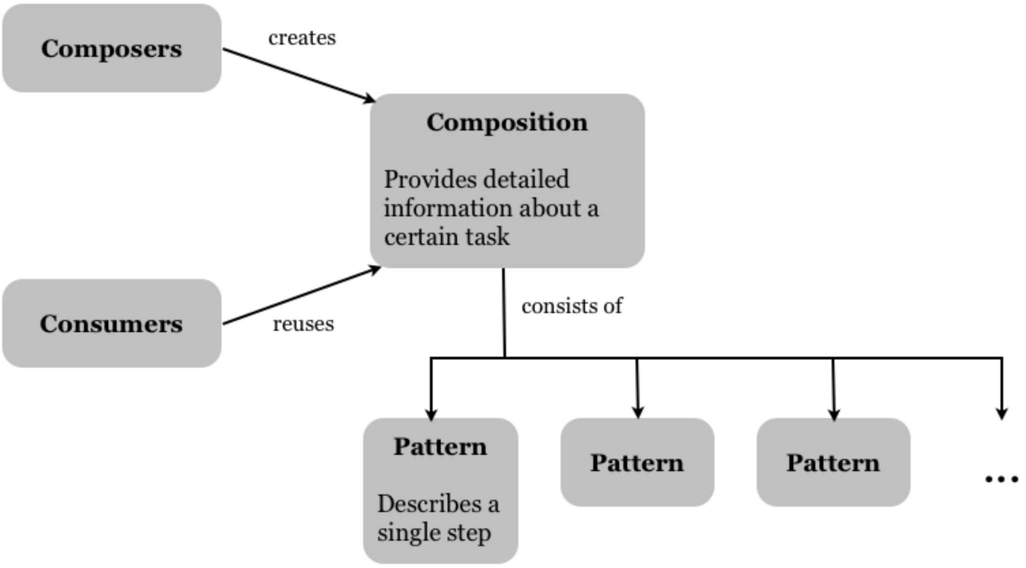
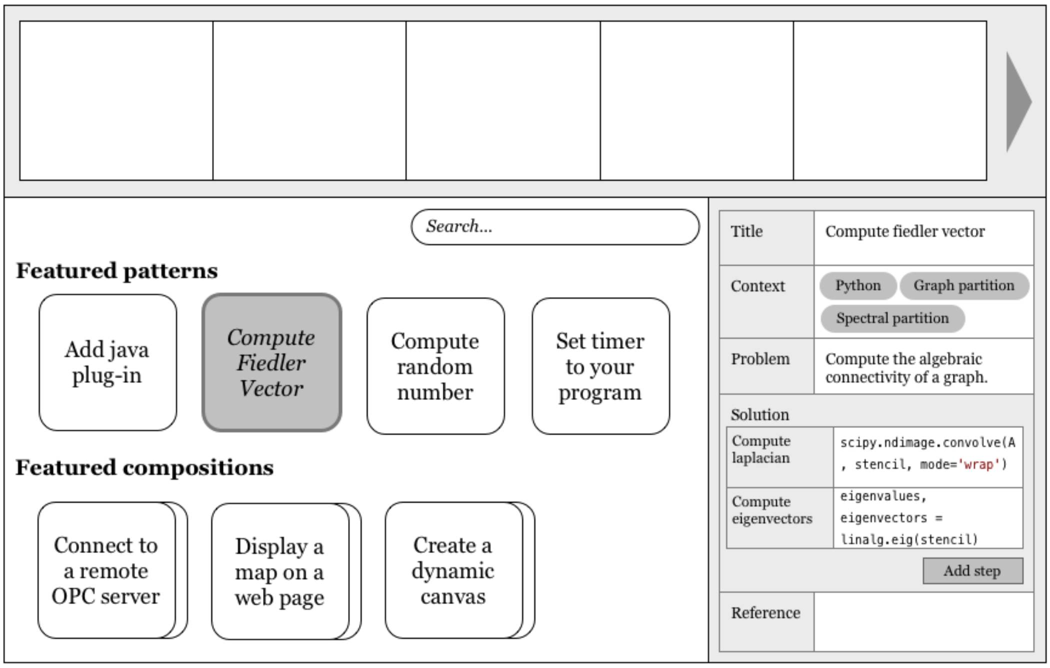


Figure 5.4: A composition now consists of multiple patterns, each representing a single step in the composition's solution.

A composition now represents a single pattern language, instead of a single pattern.

by Alexander's proposed method of using design patterns. In his book "A Pattern Language," he gave an illustration on the activity flow that he would take to build a front porch by utilizing the various design patterns included in the book. The fundamental idea is that one should address a certain task by creating their own version of a *pattern language*, which essentially is a combination of multiple patterns. He also suggested that the proper order to execute a group of patterns is to go from the "bigger" patterns that provide the general structure to the "smaller" ones that supply the details to such structure [Alexander et al., 1977]. To adjust our design to this scheme, we shift the composition scope from a single pattern to a single pattern language. In other words, a composition now consists of multiple patterns, which roughly correspond to the multiple "steps" contained by a solution in the previous design. The idea is that with such detailed information contained by a single composition, there should be even less ambiguity about the tasks that it covers.

A direct consequence of this conceptual tweak is the inevitable scope change of the information networks that we introduced in the previous design phase. Where before a



Title	Compute fiedler vector
Context	Python Graph partition Spectral partition
Problem	Compute the algebraic connectivity of a graph.

Solution

Compute laplacian	`scipy.ndimage.convolve(A, stencil, mode='wrap')`
Compute eigenvectors	`eigenvalues, eigenvectors = linalg.eig(stencil)`

Reference	

Figure 5.5: The composer application window consists of three main parts: the palette (top), the repository (bottom left), and the property panel. A selected pattern's information is displayed in an editable form in the property panel.

An information network now stands for a hierarchy of patterns, instead of compositions.

single network would depict the connections among compositions, it currently represents a hierarchy of patterns, which are slightly narrower in focus than a single composition. The reason for this change is so that we can stick to Alexander's "bigger to smaller" method of patterns tracing, a technique that will become clearer when we touch on the application form in the next section.

The Form

The consumer part of the application can be accessed from inside the IDE, while the composer part has its own application window.

The Code Mixer desktop application consists of two separate parts, one for the composers and one for the consumers. This separation is driven by the belief that each role has a different focus, and separating them would give us the possibility to tailor each part to specifically address the needs of each role. The consumer part of the application can be accessed directly from inside the IDE, under the assumption that most consumers are typically in the middle of finishing some programming task when they feel the need to conduct some information foraging. In contrast, when users have decided to build a new composition, they would be more inclined to invest some time for this activity [Wightman et al., 2012], and therefore the composer part of the application will be contained in a separate application window outside of the IDE.

The composition part of the application consists of a gallery, a palette, and a property panel.

The composer application window consists of three main parts, which can be seen in Figure 5.5. The biggest area is the *gallery*, which displays existing patterns and compositions that users can browse through. At the top of the window is the *palette*, in which users can place multiple different *patterns* to build a whole composition. The underlying idea is that a composition is akin to a step-by-step instruction that consumers should be able to follow easily. A pattern in the palette can either be directly taken from the repository, a modified version of an existing pattern, or freshly created by the users. Moreover, the palette has the ability to enforce the proper sequencing of patterns by examining their *context*. When users place a pattern in the palette, the system would inspect the context of said pattern. If the pattern lists other patterns in its context, this means that there exists "bigger" patterns that should be executed first. The system would then ask the users whether they wish to place the bigger patterns in the preceding positions in the palette.

The property panel is where the creation of a new pattern takes place.

Meanwhile, the third and last part of the composer window is the *property* panel that displays the detailed information about a pattern that is currently selected by the user (either from the gallery or the palette). Its content is similar to the composition form for composers that we proposed in our previous design. Finally, this panel is also responsible for letting the users create new patterns or modify the properties of an existing pattern and save the modified version into a new pattern.

The consumer part of the system can be accessed from inside the IDE by pressing a certain keyboard shortcut, which would evoke a pop-up window. Inside this window users can search for the right compositions, and once they find

The consumer application window is a pop-up that can be evoked by pressing a certain keyboard shortcut.

one, they can download it right away. A composition that is downloaded will be saved into a local file on the user's machine. To prevent inconsistency, the system will create a new directory for Code Mixer under the currently active code project, if one does not already exist, and save the composition file into this directory. When the composition file is opened, the content will be display in the same pop-up window, which presents the users with one pattern at a time, instead of all at once. The idea behind this is that users should focus on one step of the composition at a time, instead of distracting themselves with things that should not yet be executed (i.e. the "smaller" patterns). Users then simply need to follow the instructions in each pattern, and they can also copy the content from the solution fields of the patterns. Once copied into their own code, a comment is inserted on top of the code chunks that denote the title and the location of the composition in the local machine. On top of the consumer application window, a downloaded composition can also be opened using the composer application window, whether just to review its content or to modify its content and save it into a new composition.

The Composer Workflow

A composer builds a new compositions by placing various patterns into the palette, and they are also allowed to create new patterns as well as modify existing ones.

After opening the composer application window, users can start building a new composition right away by placing patterns in the prefered order inside the palette. When given a suggestion by the system about automatic positioning of "bigger" patterns, users can opt to skip it. To reuse existing patterns, users simply search for the appropriate ones in the gallery and drag them into the palette. Users can create a new pattern by filling out an empty form in the *property* panel, and users can also modify an existing pattern by selecting one from the gallery (or the palette) and making adjustments to its properties as desired. Once all necessary patterns have been put in the appropriate order in the palette, users can save and upload the composition into the repository. However, they first need to give the composition a proper title and, if desired, a short description of the composition itself.

Moreover, users are also allowed to modify existing compositions, which is done in a similar fashion to modifying existing patterns. Users start by either searching for an existing composition in the gallery, or simply opening a previously downloaded composition from their local file systems. Users then modify the content of the selected or opened composition by playing around with the patterns in the pallette, either by adding or removing some patterns, or by changing the sequence of the patterns. Of course, they are also given the ability to simply alter the properties of existing patterns already contained in the composition. Once finished, the new composition can be uploaded to the repository under a new title.

A composer can also modify existing compositions, both the ones in the repository or the ones in their local machine.

The Consumer Workflow

Users activate the consumer application window right from inside the IDE by pressing a certain keyboard shortcut. A pop-up window equipped with a search box will appear, and users can start searching for some relevant information right away, without ever having to leave their working environment. The search results, which contain a group of compositions, will be displayed in the sampe pop-up window, and when users click on one result, the window will display a detailed information of the selected item. Once users have decided on one item, they simply need to download the composition and this file will then be saved into the local file system.

Consumer application window is a pop-up where users can search for compositions, as well as downloading them.

A downloaded composition is then opened in the same pop-up window, and users only need to follow the sequence of patterns contained by the composition. While following the sequence, users can choose between skipping a certain pattern or integrating the content of said pattern into their own work. To verify the intention of a code block, users can take advantage of the code comments above the block in order to locate the associated composition in the local file system. This composition can then be reopened in the consumer pop-up window to browse through again, or it can be accessed through the composer application window to modify its content.

A downloaded composition file is opened in the same pop-up window, or in the composer application window

Self Assessment

The current design has increased the burden for composers, and therefore may have a hard time taking off.

In our attempt to emulate Alexander's method of utilizing design patterns, we have also inadvertently increased the burden for composers. Whereas in the previous version they are only required to fill in pattern-related information for one composition, now they have to supply similar information for each patterns contained by a single composition. We try to counter this by providing a central repository and allowing them to sample existing patterns from the repository, but this strategy relies heavily on the existing collection of patterns, and the willingness of the community to keep maintaining—and evolving—these patterns. In other words, there is a concern that the current design scheme may have a difficult time taking off, since the initial period of its life would require a lot of efforts on the users' parts. Nevertheless, we feel that the idea of a central repository is a sound one, as it will provide a stable environment for users to look for compositions and share their creations.

The current design has not improved the user experience of composers, and the linear format may be too limited.

Even though the current design has somewhat improved the interaction flow between consumers and compositions, it does little to do the same for composers. Composers are still required to leave their current work environment just to build a new composition, making it rather cumbersome when users simply wish to share some parts of their own work. Since our design scheme revolves around a thriving community, an adjustment to the system is required to allow for a more lightweight creation of compositions, so as to encourage more users to take on the role of composers. The current format for building a composition may also prove problematic, since a linearly sequenced patterns may be too restrictive a structure for some tasks. For instance, a single task may boast a few alternative solutions, and with the current design, composers are required to create a separate composition for each option.

The current design has yet to properly address the problem of code comprehension.

Finally, just like with the previous design, we have yet to properly address the problems of code comprehension. Opening the compositions in a pop-up window to read the intention behind a code block may help a little, but code comments may not be powerful enough tools to signify the existence of compositions, since they can easily get buried beneath all the other comments in the code. As a consequece, users need to remember the title of a composition before they can begin to search for its multiple instances among the plethora of comments inside the code. This problem is something that we will try to overcome in the next design phase.

5.3 Code Mixer

The first major change is to get rid of standardized forms and give users freedom in molding the shape of a composition.

At this stage, we have arrived at the realization that our initial impulse to stick as close as possible to Alexander's way of patterns may have stifled our creativity during the previous design phases, and indeed, may have caused us to violate some of our own design principles. For one, using a linear progression while building a composition may not be accommodating enough for software engineering, a field often saturated with multiple conditional cases. Furthermore, the standardized form that is utilized to structure compositions does little to remedy the problem of *visual homogeneity* that is already prevalent due to the tendency to "write" code. Indeed, the two previous designs would eventually yield a group of compositions that are visually indistinguishable from one another. As a response to this problem, the first major alteration in our design is to get rid of the standardized form and in its place, we let users mold the shapes of the compositions based on their own whims. This would reduce the generic visual quality of compositions (and thus improve their visual identities), while simultaneously taking advantage of developers' inclination to consider more free-form graphics as more "interesting" [Cherubini et al., 2007].

To achieve a more balanced community, we set out to unify the user experiences of the two roles.

Another defining characteristic of our previous designs is the segregation between the two user roles. Heretofore there has been very little overlap between the two roles beyond the fact that they both interact with compositions, and indeed, we even went so far as to present a distinctive application window for each of them. Our reasoning at the time was that each user role should be able to focus on their respective tasks with as little distractions as possible. However, the resulting design may ultimately encourage users to only identify with one particular role, as opposed to actively taking on both roles. Add in the fact that the composer role is indubitaly more demanding, and it would result in a community that is dominated by consumers. Needless to say, this is not a desirable state for our design scheme, which requires an active community that would constantly maintain and evolve the existing collection of compositions. With this in mind, in the current design phase we set out to minimize the discrepancy of labor between the two roles, as well as to unify their respective user experiences. To achieve this, all users, regardless of roles, would interact with the same part of the desktop application.

The current design also addresses code comprehension problems by using a combination of icons and associations between compositions and users code.

Perhaps the most striking tweak introduced by the current design phase is the inclusion of features that deals with code comprehension. It bears repeating that developers typically choose to gain comprehension on smaller, isolated parts of a system that are most relevant to their current task, instead of trying to understand the whole system [Singer et al., 1997, Roehm et al., 2012]. Therefore, Code Mixer would facilitate code comprehension in a more personalized way. Each developer would have their own set of aids for comprehension, depending on the *associations* between compositions and their own code that they themselves create. Each composition is assigned a unique *icon*, and whenever users specify an association between a segment of the composition with a section of their own code, the composition's icon will be inserted into the IDE. The idea of multiple icons strewn across the IDE is not unlike a bunch of colorful sticky notes that are attached to the pages of a thick report in order to provide a quick access to all the important or interesting parts of the report. Similarly, an icon that has been inserted into a certain code segment in the IDE can provide an instant access to the composition it belongs to. The composition itself would ideally present the users with enough information regarding the real world tasks that it addresses. Or, to put it concisely, Code Mixer helps code comprehension by offering a *task-oriented view* of code.

Finally, we have made the decision to revert the scope of a composition back to a single pattern. This approach would put less burden on the creation of compositions, and thus, would better support our aspiration of encouraging more users to assume the composer role. In the next section, we provide further details on how we retailor the user interface of Code Mixer to make it more effective.

The composition scope is reverted back to a single design pattern.

The Form

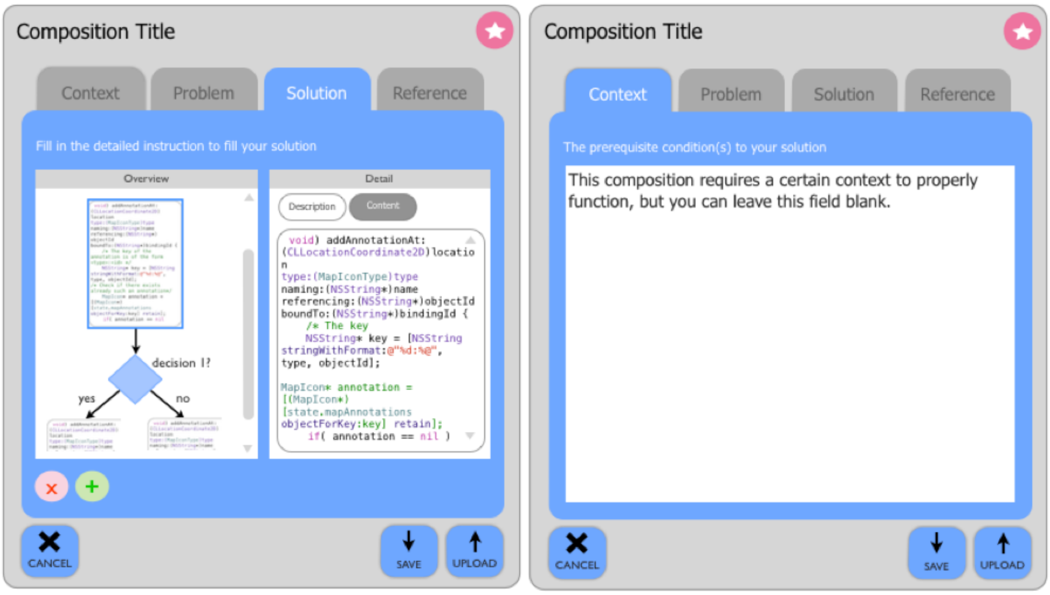
The current version of Code Mixer consists of one desktop application window, within which all interactions with compositions will take place, regardless of user roles. The application window, as can be seen in Figure 5.6, consists of four main tabs which would host the various pattern-centered components: context, problem, solution and reference. A minor change has been made regarding the contents of these components: each may now contain both textual and graphical information. To complement the four tabs, the application window would also host the title and the automatically assigned icon that we have briefly covered in the previous section, and both of these elements serve as a form of identification for the composition. It should be noted that a composition's icon is decided on a work space basis, meaning the icons do not exist in the central repository. Code Mixer keeps track of all the compositions that a code project contains, and when user downloads a composition or creates a new one, this composition will be assigned an icon that has not been used by other compositions in the project.

There is only one desktop application window, which contains a tabbed menu to host the composition.

A composition icon is decided on a work space basis.

The most interesting part of the new design is the solution tab, which serves as a blank canvas in which users can build their solutions. A solution is made out of multiple elements that are connected together to form a graph-like structure, which we would call the *solution graph*. Most people, particularly software developers, should already be familiar with building and interpreting graphs, which means that building a solution in the new design would require very little learning effort, if any. And despite their relative simplicity, graphs are flexible enough to represent a number of different scenarios, even non-linear instances such as conditional cases and loops. Indeed, Cherubini et al. [2007] pointed out

The solution tab serves as a blank canvas where users can build their own solution graphs.



```
 void) addAnnotationAt:
(CLLocationCoordinate2D)locatio
n
type:(MapIconType)type
naming:(NSString*)name
referencing:(NSString*)objectId
boundTo:(NSString*)bindingId {
    /* The key
    NSString* key = [NSString
stringWithFormat:@"%d:%@",
type, objectId];

MapIcon* annotation =
[(MapIcon*)
[state.mapAnnotations
objectForKey:key] retain];
     if( annotation == nil )
```

This composition requires a certain context to properly function, but you can leave this field blank.

Figure 5.6: The two different tabs of the Code Mixer application window. The solution canvas hosts a solution graph whose elements are the thumbnail images of their own contents.

that the paper sketches that developers produce during a brainstorming session are typically dominated by graph-like forms. Moreover, as long as users are given full spatial freedom while building their graphs, the respective personalized touches of the users would lend each of the resulting graphs a certain level of distinction, which would add to a graph's visual identity.

One major decision was to determine the form of the elements of the solution graph.

Given the prominent role that the solution graph plays in the current design, one of the major decisions taken during this design phase is to determine the visual form that the graph elements should take. We considered borrowing from the *Unified Modeling Language* (UML), as Code Mixer, after all, is marketed towards software developers. But then we quickly discovered that despite all the different types of available UML diagrams, it was almost impossible to select one particular type that is suitable to our design scheme, since most of these graphs have a very specific purpose (e.g. Use Case Diagrams model functionalities, Class Diagrams model data structures, and so forth). Code Mixer requires something that is generic enough because it is supposed to work seamlessly with a wide array of problem domains, so UML diagrams are not the best fit.

A graph element would be a thumbnail image of its own content.

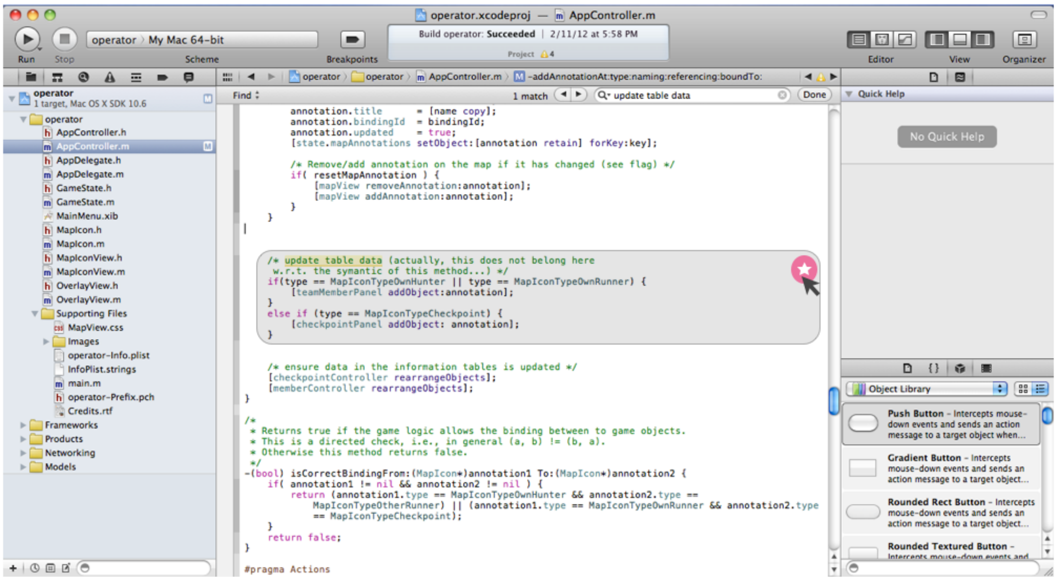
Ultimately, we draw our inspiration from one of the tools listed in this paper, Code Thumbnail [DeLine et al., 2006], specifically the way this tool creates thumbnail image representations for code files. Our idea is that a graph element should be the thumbnail image of its own content, regardless of whether it contains code chunks of conventional texts. The resulting thumbnail images would further strengthen the visual identity of a composition, which already benefits from having a distinctive graph structure. In order to prevent the thumbnail images from overflowing the screen and throwing the balance of the solution graph, a maximum width restriction is enforced on them by using the text-wrapping technique adopted from Code Bubble [Bragdon et al., 2010]. It should be reiterated that these thumbnail images are merely supposed to add to the visual identity of a composition and are therefore not meant to be read by the users.

A graph element's content would always mirror the content of the code segment that it has an association with.

A property panel in the solution tab is then provided (see Figure 5.6) so users can properly read and modify the content of a certain graph element. Each graph element has its own pair of fields, one to supply a general description and the other for the actual content. One last interesting feature of these graph elements is its capability to mirror the content of a certain code segment, provided that an *association* has been specified by the users. Namely, whenever a code segment in the IDE is modified by the users, the graph element that it has been associated with in the composition window will also transform itself, changing the appearance of its thumbnail image in the process. This should provide a certain degree of consistency and fortify the sense of connection between compositions and users' own code.

A composition icon provides instant access to a composition from the IDE.

Aside from the application window, there are also additional elements that are integrated into the code editor. We already described how a composition icon would be inserted into the IDE whenever an association is made between a code segment and a composition. The icon is added on the very first line of the code segment, and when users move the mouse cursors on top of the icon, the rel-



operator.xcodeproj — AppController.m
Run
Stop
operator › My Mac 64-bit
Scheme
Breakpoints
Build operator: Succeeded | 2/11/12 at 5:58 PM
Project 4
Editor
View
Organizer
operator › operator › AppController.m › -addAnnotationAt:type:naming:referencing:boundTo:
Find
1 match
update table data
Done
operator
1 target, Mac OS X SDK 10.6
operator
AppController.h
AppController.m
AppDelegate.h
AppDelegate.m
GameState.h
GameState.m
MainMenu.xib
MapIcon.h
MapIcon.m
MapIconView.h
MapIconView.m
OverlayView.h
OverlayView.m
Supporting Files
MapView.css
Images
operator-Info.plist
InfoPlist.strings
main.m
operator-Prefix.pch
Credits.rtf
Frameworks
Products
Networking
Models
Quick Help
No Quick Help
Object Library
Push Button - Intercepts mouse-down events and sends an action message to a target object when...
Gradient Button - Intercepts mouse-down events and sends an action message to a target object...
Rounded Rect Button - Intercepts mouse-down events and sends an action message to a target object...
Rounded Textured Button - Intercepts mouse-down events and

```
            annotation.title       = [name copy];
            annotation.bindingId   = bindingId;
            annotation.updated     = true;
            [state.mapAnnotations setObject:[annotation retain] forKey:key];

            /* Remove/add annotation on the map if it has changed (see flag) */
            if( resetMapAnnotation ) {
                [mapView removeAnnotation:annotation];
                [mapView addAnnotation:annotation];
            }
        }


    /* update table data (actually, this does not belong here
     w.r.t. the symantic of this method...) */
    if(type == MapIconTypeOwnHunter || type == MapIconTypeOwnRunner) {
        [teamMemberPanel addObject:annotation];
    }
    else if (type == MapIconTypeCheckpoint) {
        [checkpointPanel addObject: annotation];
    }

    /* ensure data in the information tables is updated */
    [checkpointController rearrangeObjects];
    [memberController rearrangeObjects];
}

/*
 * Returns true if the game logic allows the binding between to game objects.
 * This is a directed check, i.e., in general (a, b) != (b, a).
 * Otherwise this method returns false.
 */
-(bool) isCorrectBindingFrom:(MapIcon*)annotation1 To:(MapIcon*)annotation2 {
    if( annotation1 != nil && annotation2 != nil ) {
        return (annotation1.type == MapIconTypeOwnHunter && annotation2.type ==
            MapIconTypeOtherRunner) || (annotation1.type == MapIconTypeOwnRunner && annotation2.type
            == MapIconTypeCheckpoint);
    }
    return false;
}

#pragma Actions
```

Figure 5.7: When the mouse cursor hovers over the composition icon in the IDE, the relevant code segment is highlighted. Clicking on the icon would present the users with the application window containing the relevant composition.

evant code segment will be highlighted using a light gray box (see Figure 5.7). Furthermore, if users click on the icon, they will be presented with the relevant composition in the application window, and the graph element that has been associated with the selected code segment will also be highlighted.

Navigating a code using a composition window allows users to focus only on the code segments that are relevant to their current task.

Users can also quickly navigate into the other code segments in the IDE that have been associated with a composition by simply clicking on another element of the solution graph. The mouse click operation will then highlight the selected graph element, as well as make the code editor focus on the relevant code segment (even if it is located in a different code file), provided that an associated has been previously specified. This way, users can focus only on the code segments in the IDE that are relevant to their current task (which is contained by a composition), without having to be distracted by other code segments. Also, for newcomers looking to gain insight to a code project, the various icons will indicate the important or interesting parts of the code that they should focus on first. These smaller, segmented outlooks to a project would prevent the new users from becoming overwhelmed by unfamiliar pieces of code whose connections to one another are not always obvious. Finally, navigating source code by using the composition window would reprieve the users from having to constantly perform multiple textual searches on function or package names and the likes. We believe that this is a much more efficient method to traverse code than combing through a long list of search results.

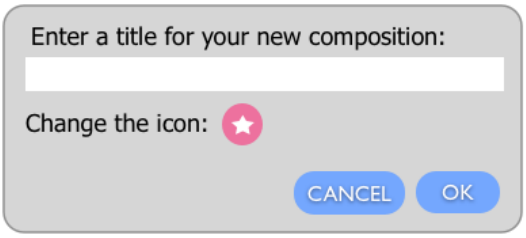
The Composer Workflow

Users no longer have to input all pattern-centered information of a composition.

To create a new composition, users calls up the Code Mixer application window by pressing a certain keyboard shortcut from anywhere in the code editor. A pop-up window materializes, bearing a blank composition, which users then fill out with as much information as needed. To ensure a lighter method of creating a composition, Code Mixer no longer requires users to supply all pattern-centered properties; only the title and the solution tab are mandatory. The are a couple of reasons behind this decision. Firstly, it was noted by Brandt et al. [2009] that when following a tutorial, users often jump directly into the various code example embedded in the tutorial and pay little attention to anything else. This finding indicates that a comprehensive set of pattern-centered information may not always be required for a composition to be truly effective.

Users are given absolute freedom to decide how detailed their compositions are and how widely they are distributed.

Furthermore, we would like to accommodate the cases in which users wish to create only the barest version of a composition. For instance, several users may collaborate on building a composition while having a discussion about a certain task. All the high-level, task-related information are exchanged orally, so they simply use Code Mixer to help them build a solution graph, and at end of the brainstorming session, each of them is given a copy of this composition for their own personal use. Perhaps one of the users will find the time to complete the composition and upload it to the central repository, or perhaps not. Code Mixer now gives the users absolute freedom as to how detailed their compositions are—and how widely they should be distributed.



Enter a title for your new composition:

Change the icon:

Figure 5.8: When users save or upload a composition, they are prompted to fill in the composition title, and are given the choice to change the composition icon.

Users can build a graph by dragging existing code segments or adding a blank element into the solution canvas.

When it comes to building a solution graph, there are several possible methods that users can use. The most common way is by combining multiple segments of existing code together, either for personal code comprehension purpose or for sharing some information with others. To do this, users simply highlight a certain code segment in the IDE that would make up an element of the solution graph, and drag it into the composition window. A thumbnail image of the selected segment would appear in the *solution canvas*, and a temporary composition icon is automatically inserted into the IDE at the first line of the selected segment (users can choose to retain this icon, or change it at any time). If a graph element requires an additional descriptive text, users can add it in the property panel (see Figure 5.6). Another way to add a graph element is by manually adding a blank element into the canvas. A blank element can contain anything from a brand new code chunk to non-code instructions (e.g., configuration parameters), or even conditional cases. It should also be noted that, to facilitate multiple alternatives for a certain solution, users are also allowed to build multiple separate graphs in a single canvas.

A composition can either be uploaded into the repository or shared using portable devices.

To finish a composition, users simply need to add one or more elements into the canvas, and connect them to one another as necessary. They are also encouraged—but not required—to fill out the information in the property panel of the solution tab, as well as the pattern-centered information in the other tabs. Once a desired composition is achieved, users can upload the composition to the repository after supplying a proper title, or they can simply save it for personal use. A composition can also be shared using portable devices such as USB sticks or emails, and it can be opened in any other machines where Code Mixer has been previously installed.

The Consumer Workflow

Consumers search for compositions in the repository, download the relevant ones, and specify associations to their own code.

Consumers of Code Mixer by definition are looking for compositions that they can reuse to tend to their own tasks. They search for the most relevant compositions by browsing the central repository, which can be done through the application window or from the web browser. Once an appropriate composition is located, users then download said composition into their local machines. A downloaded composition is then opened using the Code Mixer application window, in which its content is filtered through multiple tabs. A newly downloaded composition naturally does not yet own any association to the users' own code, and to maximize flexibility, users can specify the associations at any time they see fit. Once the associations are properly specified, users are granted the ability to efficiently navigate the important code segments, simply by traversing the solution graphs contained by the compositions that they have added to their code projects. They no longer have to memorize code file names, namespaces, packages and other similar information that could potentially overwhelm their cognitive load that should have been utilized to finish a real task.

The first method to specify a composition is by dragging a graph element into the code editor.

There are two possible methods to create an association between a composition and users' existing work. The first method is by dragging an element from the solution graph into the code editor. This is akin to the typical "copy & paste" activity that most developers should already be familiar with. The content of the graph element will then be copied into a section in the code editor where the mouse cursor is currently located. An automatic composition icon will be added into the first line of the copied code segment, and when the mouse cursor hovers over this icon, the whole code segment will be highlighted using a light gray box. Moreover, when the code segment is modified, the change will be reflected in the content of the associated graph element, as well as its corresponding thumbnail image.

The second method to specify a composition is by dragging a segment from the code editor into a certain graph element in the solution canvas.

The second method to create an association works in a very similar fashion, but is also fundamentally different: it goes in the exact opposite direction. This method is intended for the cases in which an association is made between a composition and something that already exist in users' own work (i.e. they have previously finished some steps required by a composition). To specify the association, users simply highlight a certain segment of their own code and drag it into a certain element of the solution graph. Just like in the previous method, a composition icon will be inserted in code editor at the first line of the selected area. However, in this case Code Mixer will also adjust the content of the selected graph element to mirror users' existing work. Notice that this method is very similar to the composer workflow, the only difference being that users do not add a new element into the solution canvas, but instead modify an existing one. It should be noted that not all graph elements have to be associated with user's own work. This flexibility is particularly important since, as we have stated before, not all elements would contain code chunks.

Since everything is done in the same application window, users can switch between roles instantly.

Finally, a consumer can now switch role into a composer almost instantly, and vice versa. When users open a downloaded composition and specify some associations, they take on the role of a consumer. However, once they start to modify the existing content of a composition, either by adding new elements into the solution graph or by modifying the pattern-centered information of a composition, they are already assuming the role of a composer. Throughout the work session users can keep switching back and forth between the two roles easily, since everything is done in the same application window. Moreover, when a newly modified composition is uploaded back into the repository, it will then be saved as a new composition, keeping the original version in the repository. Ultimately, the unified user experience has successfully erased the separation between the two user roles, and this would in turn encourage more users to play a more active role within the Code Mixer environment.

Self Assessment

Our current design scheme has managed to unify the user experiences of both user roles, to the point that they are almost interchangeable. When a "composer" creates a brand new graph element and proceeds to drag said element into the code editor, she is actually performing a consumer-related task. Similarly, when a "consumer" modifies the content of a previously downloaded composition and uploads it back into the repository, he is actually taking on the role of composer. Furthermore, we also reduce the discrepany of workload between the two roles by taking away the requirement of building only complete, pattern-approved compositions. We believe that these characteristics would eventually help us to achieve Alexander's vision of "letting patterns evolve" and ensure that we achieve a thriving user community that has been our goal from the start.

We have managed to offer seamless transitions between user roles and reduce the discrepancy of workload between the two.

Our current design can also help users with their code comprehension endeavors, with the caveat that proper associations have been specified between users' code and the compositions. By taking advantage of the composition icons scattered across a project, user can instantly gain insight of the intention of a certain code segment. Morever, by using the solution graphs, users can jump across a code project without ever losing grasp of the composition that they are focusing on. And since each composition revolves around a single task, we therefore provide the users with a task-oriented view of their code.

We provide task-oriented view of code by using the composition icons and the solution graphs.

However, we have also noticed some portions of the design that could benefit from a little more polish. Perhaps the most glaring oversight is how restrictive the current interface turns out to be, particularly for the solution canvas, which essentially is the most important part of a composition. On top of having to share the window space with interface elements such as tabs and buttons, a solution canvas also has to fight for space with the property panel, leaving a very limited area for users to build and actually see their solution graphs. Since we have established that components such as context and references are no longer required in or-

The current user interface is still not as efficient as it could be.

der to submit a composition, they no longer need to occupy the same level of space as the solution canvas. In the final section of the design phase, we will perform some further tweaks to the user interfaces to make Code Mixer as efficient and user-friendly as possible.

5.4 Code Mixer: Retouched

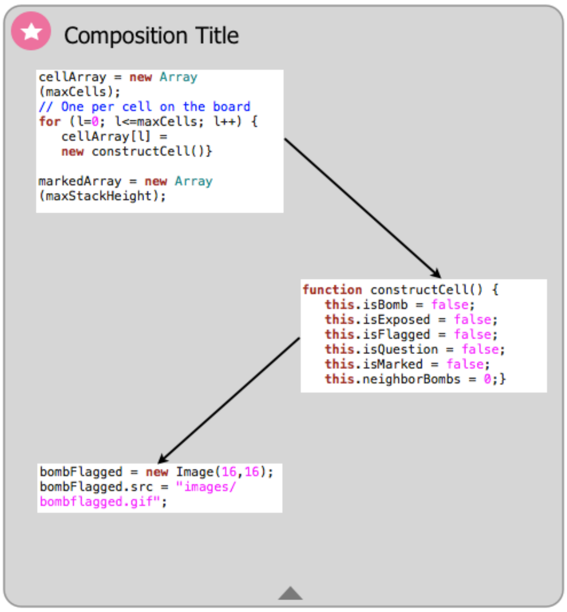
The last iteration of our design phases is primarily aimed at fine-tuning the user interface of Code Mixer. Our major inspiration is the work of Tufte and Graves-Morris [1983], which encourages designs that have been trimmed of as much "fats" as possible in order to provide the most efficient use of space for the actual content. However, in the end the application would retain most of the essence of the previous design.

The Form

The whole application window now serves as the solution canvas.

We have established that the solution graph is the most important aspect of a composition, since most developers would rather focus on the various code chunks rather than any other type of information [Brandt et al., 2009]. Accordingly, our very first focus is to provide as much room as possible for the solution canvas, and to achieve this, we simply wipe away almost everything else from the application window. As can be seen from Figure 5.9, the whole Code Mixer application window now essentially serves as the solution canvas. The only other visible elements are the composition title and icon, which are both placed at the top of the window. Meanwhile, a set of menu are hidden at the bottom of the window, and will be displyed when the mouse cursor hovers over the bottom area of the application window. The interface should now give users ample room to build and play around with the structure of their solution graphs.

Even though they are no longer mandatory components, a composition's pattern-centered properties should remain



```
cellArray = new Array
(maxCells);
// One per cell on the board
for (l=0; l<=maxCells; l++) {
   cellArray[l] =
   new constructCell()}

markedArray = new Array
(maxStackHeight);
```

```
function constructCell() {
   this.isBomb = false;
   this.isExposed = false;
   this.isFlagged = false;
   this.isQuestion = false;
   this.isMarked = false;
   this.neighborBombs = 0;}
```

```
bombFlagged = new Image(16,16);
bombFlagged.src = "images/
bombflagged.gif";
```

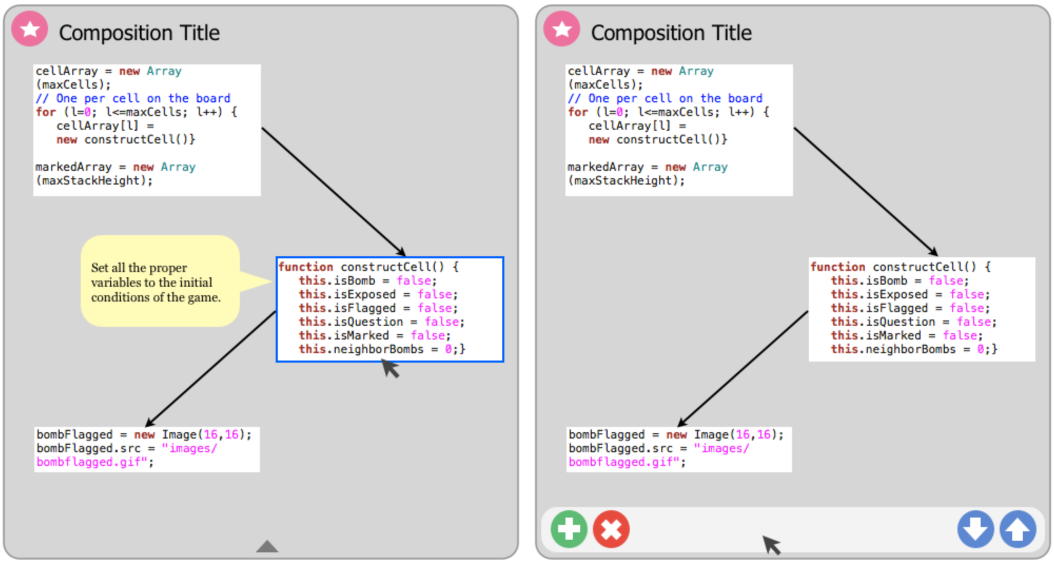
Figure 5.9: The whole application window serves as the solution canvas.

in play, since they offer a detailed view into the actual task that is addressed by the composition. And since they have been removed from the default interface, our next focus is then to find a proper way to display these components. Ultimately it was decided that they should be hosted by a new application menu, which, when activated, would push forward a panel containing several editable fields, each corresponds to a single pattern element. This approach is actually quite prevalent and has been utilized by a lot of popular appications (e.g., Microsoft Words boasts a wide array of configurable elements that are attached in the menu), so users should be relatively unfazed by this change. The structure of the information itself remains unchanged, and each field should still be able to handle both texts and graphical objects.

The pattern-centered information are now included in the application menu.

Another component that has lost its place in the new interface is the property panel of the solution canvas. This panel is a supporting act to the solution graph, and therefore it should not be grouped in the menu with the other pattern-centered information. As an alternative, each graph ele-

Each graph element has its own description panel which is hidden by default.



```
cellArray = new Array
(maxCells);
// One per cell on the board
for (l=0; l<=maxCells; l++) {
   cellArray[l] =
   new constructCell()}

markedArray = new Array
(maxStackHeight);
```

Set all the proper variables to the initial conditions of the game.

```
function constructCell() {
   this.isBomb = false;
   this.isExposed = false;
   this.isFlagged = false;
   this.isQuestion = false;
   this.isMarked = false;
   this.neighborBombs = 0;}
```

```
bombFlagged = new Image(16,16);
bombFlagged.src = "images/
bombflagged.gif";
```

```
cellArray = new Array
(maxCells);
// One per cell on the board
for (l=0; l<=maxCells; l++) {
   cellArray[l] =
   new constructCell()}

markedArray = new Array
(maxStackHeight);
```

```
function constructCell() {
   this.isBomb = false;
   this.isExposed = false;
   this.isFlagged = false;
   this.isQuestion = false;
   this.isMarked = false;
   this.neighborBombs = 0;}
```

```
bombFlagged = new Image(16,16);
bombFlagged.src = "images/
bombflagged.gif";
```

Figure 5.10: When the mouse cursor hovers over a graph element, the description panel appears (left). When the mouse cursor hovers over the bottom of the application window, a set of menu appears. The left menu group deals with graph elements, while the right menu group is for downloading and uploading a composition.

ments now boasts its own invisible panel, which will be displayed when mouse cursor hovers on top of the element. User can additionally modify the content of this panel by either using a certain keyboard shortcut of by selecting the approapriate menu that can be accessed through mouse right click. One last change to this panel is the fact that it no longer hosts a readable version of a graph element, and is supposed to only contain a description of the graph element to help users understand its intentions.

Users can now zoom in and out of the canvas, as well as pan around it.

Naturally, the next thing that we have to focus on is to provide the users with the ability of reading the content of the solution graph properly. The eventual technique we employ is inspired by Code Canvas [DeLine and Rowan, 2010] and Shrimp Views [Storey and Muller, 1995], which is to allow users to zoom in and out of the canvas as they please. When a composition is first opened, by default the entirety of the solution graph is displayed, fitted to the current size of the application window. To get a closer look at the content of an element, users zoom in as far in as needed, or, if an association has been specified, they can also click the element and see its content in the code editor. To add more convenience, users are now also allowed to pan around the solution canvas to inspect the other elements. And when they want to view an overall structure of a graph, they can simply zoom as far back out as desired. It should be noted that the graph elements' hidden panels are not affected by the level of zoom in the canvas, they remain readable at all times.

The User Workflow

The interaction styles mostly remain the same, with some minor adjustments.

No drastic change has been implemented in the way users interact with the system, the details of which can be seen in the previous design phase (see Section 5.3 in p. 73). The fundamentals are still the same, although the adjustments in the user interface ultimately modify some of the interaction details as well. For one, when users drag a code segment from the code editor to the solution canvas, which will add a new element to the graph, the property panel of said element will also be displayed, so users can directly fill in the description of said element, if they wish to do so. Of course, they can always add or modify this information at any other time, by using the correct menu (which has been described the previous section). Additionally, when users would like to supply the pattern-centered information of a composition, they have to evoke the appropriate panel from the application menu, and then they can proceed to input as much information as necessary. Similarly, when users are taking a more passive role with the compositions, all interactions remain the same except that they have to hover the mouse cursor on top of a graph element to see the element's description and evoke the appropriate menu to see the pattern-centered information.

Self Assessment

As this is the final design phase, we are bringing back all the design principles that have been established in Chapter

4, in order to determine how effectively our design fulfills each point.

The solution graph is the most important element in our design.

Visual Emphasis. Our design does put more importance on the visual representation of a composition than its textual information. Indeed, the whole application window serves as a canvas for the user-generated solution graphs, while text-based information (pattern-centered information and graph element descriptions) is relegated to the background. There are also various measures to prevent visual uniformity, namely the composition icons, the user-based structures of the solution graphs, and the content-based thumbnails of the graph elements.

Users can create a composition just by drag-and-drop, and it can be shared through a single click.

Portability. Our design lets user build compositions without having to leave their code editor, and solution graphs can be constructed merely by dragging and dropping already-existing code segments into the canvas. Moreover, only the solution graph and the composition title are required, giving the users the freedom to make a composition as meticulous as they desire it to be. Meanwhile, users can upload their newly created or modified compositions into the central repository with a single click, and they are also allowed to distribute the compositions using portable devices and even emails. The central repository itself serves as a stable environment that spares the users from having to comb through the vastness of the Web.

Users can access the task-related information of a code segment using a single click.

Task Oriented. Our design enforces this principle by making sure that a composition revolves around a single task. As a matter of fact, the root of a composition is a real world task whose solution users describe in the form of a graph. Users can gain insight into the purpose of a certain code segment just by clicking the composition icon that has been attached onto the code editor, and they are instantly presented with the relevant composition, which again, revolves around a single task. In other words, users are only a click away from gaining insight of the task that is addressed by a certain code segment. The only caveat is that the effectiveness of this approach heavily relies on how meticulous users are when building the compositions and specifiying the associations.

Accessibility. All elements of our design are decidedly intented to be as conventional as possible in order to reduce the learning effort. It is a safe assumption that most users (who are mostly software developers) should already be familiar with interpreting a graph, as well as the *drag-and-drop* and *zoom-and-pan* interaction styles. Relegating the pattern-centered information into the application menu is also an approach that has been utilized by many popular applications, so users should already be relatively familiar with it.

Elements such as graphs, drag-and-drop and zoom-and-pan interaction styles are all already quite prevalent in the real world.

Modifiability. This property is closely related to our attempt at blurring the line between the two user roles. Indeed, users can easily modify a composition that is created by others, either by altering the solution graph or the pattern-centered information, since everything is done in one application window. The newly modified composition can either be uploaded back into the repository, distributed using portable devices, or saved into the local file system for users' own personal use.

Users can easily modify a composition that is created by others, since everything is done in the same window.

Collaboration-Friendly. Users can brainstorm together and collaborate in creating a composition, not unlike how they would work together using mediums such as the whiteboard. However, our current design only facilitates the creation of a composition at a single machine at a time. Users from remote locations can also work on a composition together, but they are required to send the composition back and forth between each other and modify it separately.

Users can collaborate on creating a composition, but it has to be done in a single machine.

Spatial Freedom. The fundamental idea of the solution canvas is that users are given full flexibility in determining the structure of their solution graphs. They can place any graph elements at any points on the canvas, and they are also allowed to connect as many elements as desired to a graph, or even to create multiple graphs to represent alternative solutions.

Users are given full flexibility in determining the structure of their solution graphs.

Recordability. Everytime a user save a composition or upload it to the repository, it automatically creates a record of said composition. When a composition is modified by other users, they are not going to be overridden, but instead the modified composition will be saved into a new one.

The repository keeps track of every composition that has been uploaded.

Ultimately, it has been shown that our design has in fact managed to fulfill all of our design principles, although some principles are fulfilled more effectively than the others. It does not prove, however, that our design already offers an effective method to help developers solve their various code comprehension and information foraging problems. To get a better idea of this, we conducted some user studies whose details will be presented in the next chapter.

Chapter 6

Evaluation

In order to acquire a better idea of where we should carry our design in the future, we present the latest version of Code Mixer to a group of users. Since our main concern does not lie in implementing the actual system, we built an HTML-based simulated environment for the users to try out the system, and we conducted a set of *qualitative user studies* with subjects who fall within our target market: developers. The results of the study were quite encouraging, and some of the user inputs were ultimately incorporated into the final design of Code Mixer.

6.1 Our Initial Hypotheses

A set of hypotheses, based on our design decisions, is established.

To kick off this evaluation phase, we establish a set of assumptions that we have made during the design period. We do this so we can verify the effectiveness of our design, and to easily see which areas would require further adjustments based on the reactions of the users. The following is a list of items that we would like to examine:

1. Users can distinguish one composition from the others using a combination of its icons, its titles, and its visual identity

2. The structure of the solution graph and its content-based graph elements provide strong enough visual identity to a composition

3. Giving users full flexibility when building the solution graph will result in wide variety of graph structures

4. The actual content of compositions can be easily understandable, and can be used to determine the most relevant item from a list of search results

5. Users would have no problem navigating an unfamiliar code base by traversing the solution graphs.

6.2 The Prototype

The prototype is a simulated environment built using HTML, CSS and JavaScript.

The prototype that we built is not a fully functional rendering of our design, but rather a simulated environment that exhibits most of the features and interaction styles that have been established during the design period. The simulation itself should not require too much effort to build, since there is a possibility that a radical twist to the overall design might be required as a result of the studies. Ultimately, we reached a decision to build an HTML-based prototype, since it is among the simplest and most flexible platforms there is. This is then further supported by CSS (*Cascading Style Sheets*) to enhance the visual quality of the prototype and JavaScript to lend some dynamic capabilities.

A web version of Minesweeper is used to provide the content to the simulated environment.

Before actually building the prototype, a set of working code is needed to provide the "content" of the simulated environment. This code should not be too long so as not to overwhelm the users, but should also exhibit a certain level of complexity so we can verify the effectiveness of our design in assisting code comprehension. After considering several candidates, a web version of the game *Minesweeper* is eventually selected as the main subject of the prototype. The reason for choosing Minesweeper is because it is a standard game that most people are familiar with, so the study participants would be able to really focus on understanding how the code is structured without having to wonder about the expected behavior of executing the code.

Furthermore, we choose a web version of the game due to the relative popularity (and simplicity) of HTML and JavaScript that are used to build this version of the game. It is preferable that study participants are familiar with the programming language presented during the study, otherwise they would not be able to solve the tasks that are going to be included. We borrowed the Minesweeper code from http://www.chezpoor.com[1] , and the code consists of three different files and boasts around 1,000 lines of code, which is an appropriate size that suits our purpose. Based on this code, we built seven different compositions (see Appendix A), involving an array of differing functionalities (e.g. selecting a certain menu item, clicking a certain button, and so forth). These compositions will be presented to the users during the study to evaluate how effective they are in helping users understand unfamiliar code.

Seven different compositions are built out of the code, so we can see how effective they are in helping users understand the code.

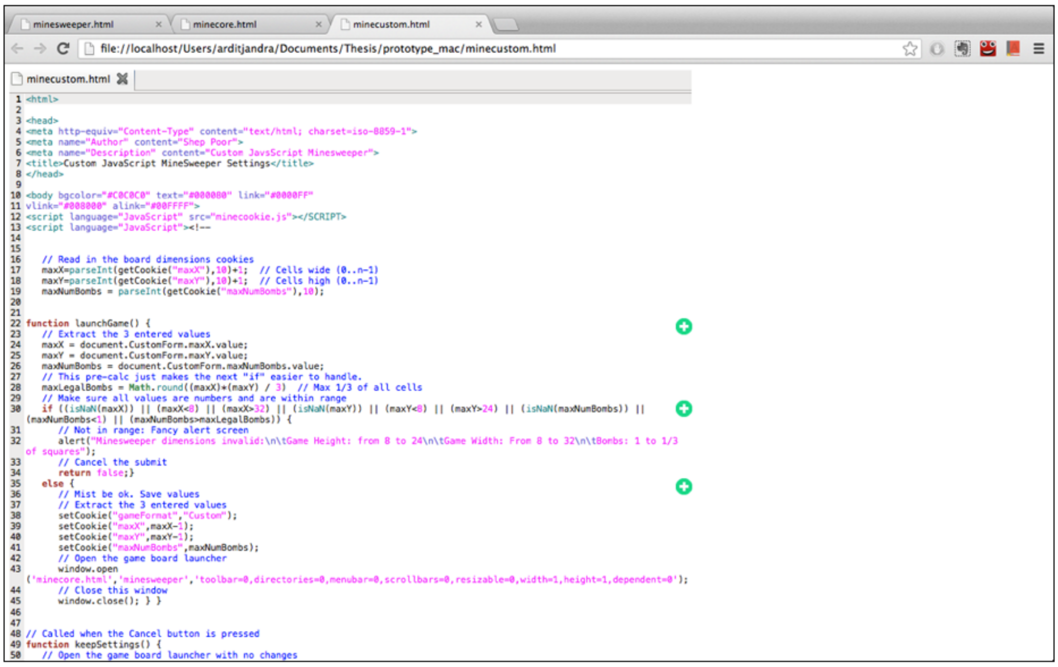
To simulate a code editor, three different HTML files are built, each corresponds to one file of the Minesweeper code. Each HTML file essentially consists of a group of screen shots of the code that are taken from a separate code editor. These screen shots are then stacked together on an empty HTML page until it looks exactly like a typical code editor, except that the users cannot do anything to its content (since they really are images). The resulting three HTML files are then opened simultaneously in a web browser using multiple browser tabs, in order to mimic the tabbed views of a typical code editor. This would then serve as the foundation to our simulated environment (see Figure 6.1).

The foundation of the simulated environment are HTML pages with screen shots of code files.

The final step in building the prototype is to integrate the pre-made compositions—and some interactivity—into the simulated environment. CSS is utilized to position and style all of the design details and objects, which include the composition window and its various content such as the composition title, the solution graph and the descriptive panels of the graph elements (see Figure 6.2). The graph

CSS and Javascript are used to integrate some compositions and interactivity into the simulated environment.

[1] http://www.chezpoor.com/minesweeper/minesweeper.html

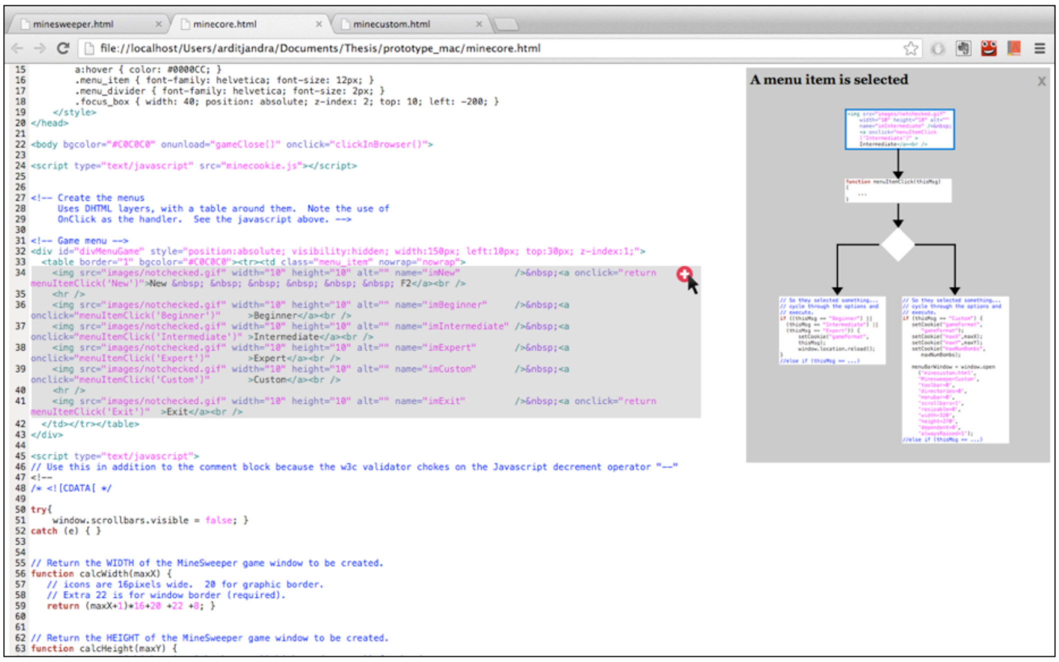

minesweeper.html
minecore.html
minecustom.html
file://localhost/Users/arditjandra/Documents/Thesis/prototype_mac/minecustom.html
minecustom.html

```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="Author" content="Shep Poor">
<meta name="Description" content="Custom JavsScript Minesweeper">
<title>Custom JavaScript MineSweeper Settings</title>
</head>

<body bgcolor="#C0C0C0" text="#000080" link="#0000FF"
vlink="#008000" alink="#00FFFF">
<script language="JavaScript" src="minecookie.js"></SCRIPT>
<script language="JavaScript"><!--


   // Read in the board dimensions cookies
   maxX=parseInt(getCookie("maxX"),10)+1;  // Cells wide (0..n-1)
   maxY=parseInt(getCookie("maxY"),10)+1;  // Cells high (0..n-1)
   maxNumBombs = parseInt(getCookie("maxNumBombs"),10);


function launchGame() {
   // Extract the 3 entered values
   maxX = document.CustomForm.maxX.value;
   maxY = document.CustomForm.maxY.value;
   maxNumBombs = document.CustomForm.maxNumBombs.value;
   // This pre-calc just makes the next "if" easier to handle.
   maxLegalBombs = Math.round((maxX)*(maxY) / 3)  // Max 1/3 of all cells
   // Make sure all values are numbers and are within range
   if ((isNaN(maxX)) || (maxX<8) || (maxX>32) || (isNaN(maxY)) || (maxY<8) || (maxY>24) || (isNaN(maxNumBombs)) ||
(maxNumBombs<1) || (maxNumBombs>maxLegalBombs)) {
      // Not in range: Fancy alert screen
      alert("Minesweeper dimensions invalid:\n\tGame Height: from 8 to 24\n\tGame Width: From 8 to 32\n\tBombs: 1 to 1/3
of squares");
      // Cancel the submit
      return false;}
   else {
      // Mist be ok. Save values
      // Extract the 3 entered values
      setCookie("gameFormat","Custom");
      setCookie("maxX",maxX-1);
      setCookie("maxY",maxY-1);
      setCookie("maxNumBombs",maxNumBombs);
      // Open the game board launcher
      window.open
('minecore.html','minesweeper','toolbar=0,directories=0,menubar=0,scrollbars=0,resizable=0,width=1,height=1,dependent=0');
      // Close this window
      window.close(); } }


// Called when the Cancel button is pressed
function keepSettings() {
   // Open the game board launcher with no changes
```

Figure 6.1: The simulated environment runs on a web browser, and is designed to look like a typical code editor. The composition icons are visible across the "code editor."

elements themselves are made out of screen shots of relevant code chunks that have been taken from a separate code editor. Meanwhile, on the "code editor" part, CSS is also responsible for placing the composition icons and the gray box that highlights the relevant code segment. We further utilized CSS to "turn off" the elements that are not displayed by default (i.e. the highlight boxes, the composition window), and ultimately it is just a matter of writing some JavaScript functions that would display and hide certain elements based on users' interactions with the prototype. And finally, JavaScript also helps to provide the ability to change the focus of the "code editor" depending on the selected graph element.

In the end, we have a web browser-based environment that manages to simulate most of the features that we have proposed in our design of Code Mixer. Everything is as has been described in the previous chapter: clicking on a icon in the editor will display the relevant composition in the ap-


minesweeper.html
minecore.html
minecustom.html
file://localhost/Users/arditjandra/Documents/Thesis/prototype_mac/minecore.html
A menu item is selected
X

```
        a:hover { color: #0000CC; }
        .menu_item { font-family: helvetica; font-size: 12px; }
        .menu_divider { font-family: helvetica; font-size: 2px; }
        .focus_box { width: 40; position: absolute; z-index: 2; top: 10; left: -200; }
    </style>
</head>

<body bgcolor="#C0C0C0" onunload="gameClose()" onclick="clickInBrowser()">

<script type="text/javascript" src="minecookie.js"></script>


<!-- Create the menus
     Uses DHTML layers, with a table around them.  Note the use of
     OnClick as the handler.  See the javascript above. -->

<!-- Game menu -->
<div id="divMenuGame" style="position:absolute; visibility:hidden; width:150px; left:10px; top:30px; z-index:1;">
  <table border="1" bgcolor="#C0C0C0"><tr><td class="menu_item" nowrap="nowrap">
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imNew"          />&nbsp;<a onclick="return
menuItemClick('New')">New &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; F2</a><br />
    <hr />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imBeginner"     />&nbsp;<a
onclick="menuItemClick('Beginner')"      >Beginner</a><br />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imIntermediate" />&nbsp;<a
onclick="menuItemClick('Intermediate')" >Intermediate</a><br />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imExpert"       />&nbsp;<a
onclick="menuItemClick('Expert')"        >Expert</a><br />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imCustom"       />&nbsp;<a
onclick="menuItemClick('Custom')"        >Custom</a><br />
    <hr />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imExit"         />&nbsp;<a onclick="return
menuItemClick('Exit')"  >Exit</a><br />
  </td></tr></table>
</div>

<script type="text/javascript">
// Use this in addition to the comment block because the w3c validator chokes on the Javascript decrement operator "--"
<!--
/* <![CDATA[ */

try{
    window.scrollbars.visible = false; }
catch (e) { }


// Return the WIDTH of the MineSweeper game window to be created.
function calcWidth(maxX) {
   // icons are 16pixels wide.  20 for graphic border.
   // Extra 22 is for window border (required).
   return (maxX+1)*16+20 +22 +8; }


// Return the HEIGHT of the MineSweeper game window to be created.
function calcHeight(maxY) {
```

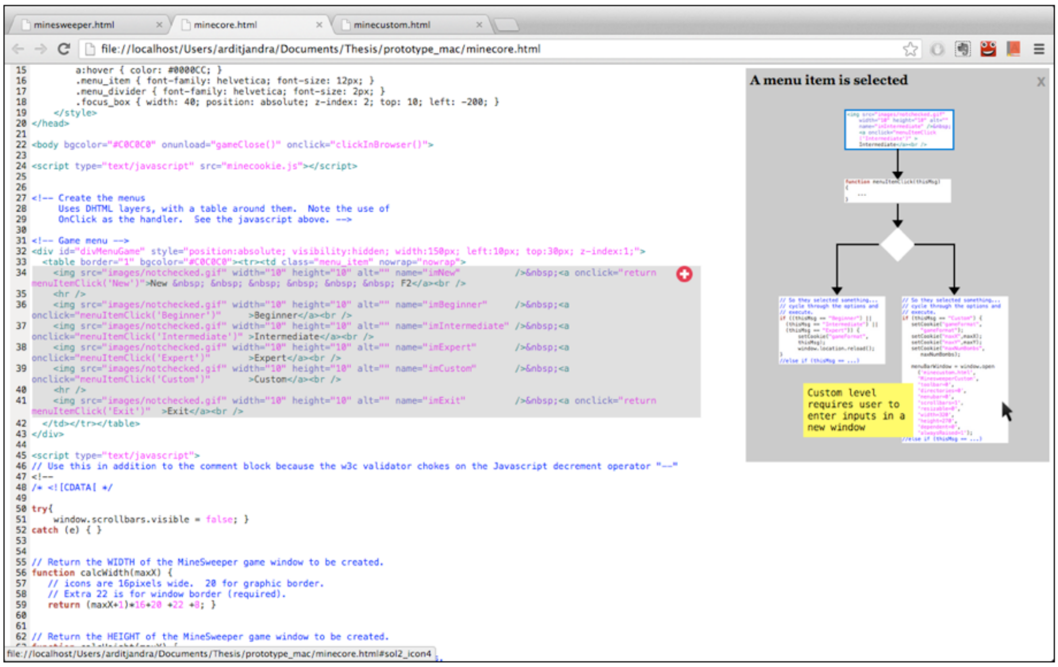
```
<img src="images/notchecked.gif"
    width="10" height="10" alt=""
    name="imIntermediate" />&nbsp;
    <a onclick="menuItemClick
    ('Intermediate')" >
    Intermediate</a><br />
```

```
function menuItemClick(thisMsg)
{
    ...
}
```

```
// So they selected something...
// cycle through the options and
// execute.
if ((thisMsg == "Beginner") ||
  (thisMsg == "Intermediate") ||
  (thisMsg == "Expert")) {
      setCookie("gameFormat",
      thisMsg);
      window.location.reload();
}
//else if (thisMsg == ...)
```

```
// So they selected something...
// cycle through the options and
// execute.
if (thisMsg == "Custom") {
   setCookie("gameFormat",
      "gameFormat");
   setCookie("maxX",maxX);
   setCookie("maxY",maxY);
   setCookie("maxNumBombs",
      maxNumBombs);

   menuBarWindow = window.open
     ('minecustom.html',
     'MinesweeperCustom',
     'toolbar=0',
     'directories=0',
     'menubar=0',
     'scrollbars=1',
     'resizable=0',
     'width=320',
     'height=270',
     'dependent=0',
     'alwaysRaised=1');
//else if (thisMsg == ...)
```

Figure 6.2: Clicking the composition icon will display the application window.


minesweeper.html
minecore.html
minecustom.html
file://localhost/Users/arditjandra/Documents/Thesis/prototype_mac/minecore.html
A menu item is selected
X

```
        a:hover { color: #0000CC; }
        .menu_item { font-family: helvetica; font-size: 12px; }
        .menu_divider { font-family: helvetica; font-size: 2px; }
        .focus_box { width: 40; position: absolute; z-index: 2; top: 10; left: -200; }
    </style>
</head>

<body bgcolor="#C0C0C0" onunload="gameClose()" onclick="clickInBrowser()">

<script type="text/javascript" src="minecookie.js"></script>


<!-- Create the menus
     Uses DHTML layers, with a table around them.  Note the use of
     OnClick as the handler.  See the javascript above. -->

<!-- Game menu -->
<div id="divMenuGame" style="position:absolute; visibility:hidden; width:150px; left:10px; top:30px; z-index:1;">
  <table border="1" bgcolor="#C0C0C0"><tr><td class="menu_item" nowrap="nowrap">
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imNew"          />&nbsp;<a onclick="return
menuItemClick('New')">New &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; &nbsp; F2</a><br />
    <hr />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imBeginner"     />&nbsp;<a
onclick="menuItemClick('Beginner')"      >Beginner</a><br />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imIntermediate" />&nbsp;<a
onclick="menuItemClick('Intermediate')" >Intermediate</a><br />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imExpert"       />&nbsp;<a
onclick="menuItemClick('Expert')"        >Expert</a><br />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imCustom"       />&nbsp;<a
onclick="menuItemClick('Custom')"        >Custom</a><br />
    <hr />
    <img src="images/notchecked.gif" width="10" height="10" alt="" name="imExit"         />&nbsp;<a onclick="return
menuItemClick('Exit')"  >Exit</a><br />
  </td></tr></table>
</div>

<script type="text/javascript">
// Use this in addition to the comment block because the w3c validator chokes on the Javascript decrement operator "--"
<!--
/* <![CDATA[ */

try{
    window.scrollbars.visible = false; }
catch (e) { }


// Return the WIDTH of the MineSweeper game window to be created.
function calcWidth(maxX) {
   // icons are 16pixels wide.  20 for graphic border.
   // Extra 22 is for window border (required).
   return (maxX+1)*16+20 +22 +8; }


// Return the HEIGHT of the MineSweeper game window to be created.
```

```
<img src="images/notchecked.gif"
    width="10" height="10" alt=""
    name="imIntermediate" />&nbsp;
    <a onclick="menuItemClick
    ('Intermediate')" >
    Intermediate</a><br />
```

```
function menuItemClick(thisMsg)
{
    ...
}
```

```
// So they selected something...
// cycle through the options and
// execute.
if ((thisMsg == "Beginner") ||
  (thisMsg == "Intermediate") ||
  (thisMsg == "Expert")) {
      setCookie("gameFormat",
      thisMsg);
      window.location.reload();
}
//else if (thisMsg == ...)
```

```
// So they selected something...
// cycle through the options and
// execute.
if (thisMsg == "Custom") {
   setCookie("gameFormat",
      "gameFormat");
   setCookie("maxX",maxX);
   setCookie("maxY",maxY);
   setCookie("maxNumBombs",
      maxNumBombs);

   menuBarWindow = window.open
     ('minecustom.html',
     'MinesweeperCustom',
     'toolbar=0',
     'directories=0',
     'menubar=0',
     'scrollbars=1',
     'resizable=0',
     'width=320',
     'height=270',
     'dependent=0',
     'alwaysRaised=1');
//else if (thisMsg == ...)
```

Custom level requires user to enter inputs in a new window

file://localhost/Users/arditjandra/Documents/Thesis/prototype_mac/minecore.html#sol2_icon4

Figure 6.3: Placing the mouse cursor over a graph element displays its description panel.

The prototype includes most of the proposed features, although some features are omitted due to technical limitations.

plication window, hovering on a graph element will bring up the elemement's description (see Figure 6.3), and clicking a graph element will make the code editor focus on the associated code segment. However, some features are omitted from the prototype due to the limited technical capabilities of our selected technologies. The unavailable features include the ability to modify a composition, the *zoom-and-pan* and *drap-and-drop* interaction styles, and the pattern-centered information of the compositions. The last feature is ultimately removed from the prototype because it takes a lot of effort to build one proper, pattern-approved composition, let alone seven, which actually confirms our own suspicion from the earlier design phases. Nevertheless, these omissions should not actively prevent us from verifying the hypotheses that we have presented in the earlier part of this chapter (see Section 6.1 in p. 91).

6.3 User Studies

Qualitative user studies were used to gauge the potential of our design.

We chose to conduct some *qualitative user studies* to explore the idea of using user-generated solution graphs to help code comprehension and distribution. Accordingly, it is not a big concern of ours to measure the accuracy of the solutions that users would supply, nor the time it takes them to finish the tasks. Instead, we would like to gauge the potential of our design scheme, and therefore, user feedbacks, mainly those regarding the experience of using the prototype, are of utmost importance.

All users in the study are familiar with HTML, JavaScript, and the Minesweeper game.

The user studies were conducted with a total of 10 participants, all of which have at least moderate experience working with HTML and JavaScript. All participants are also well aware of the purpose and the available interactions of the Minesweeper game, if not necessarily how to play it. These prerequisites are necessary so that during the study, users can fully focus on understanding how the code is written to achieve the desired result.

Protocol

At the start of the study sessions, users are given a brief explanation on the general purposes of our design, as well as the emphasis that in the study, a simulated environment will be presented instead of a functional version of the system. To ease them into the proper mindset, users are asked to imagine themselves as a new member of a development team who has been asked to learn a new code project. Users are then allowed to familiarize themselves with the Minesweeper code and the simulated environment by scanning through the "code editor" and playing around with its various features.

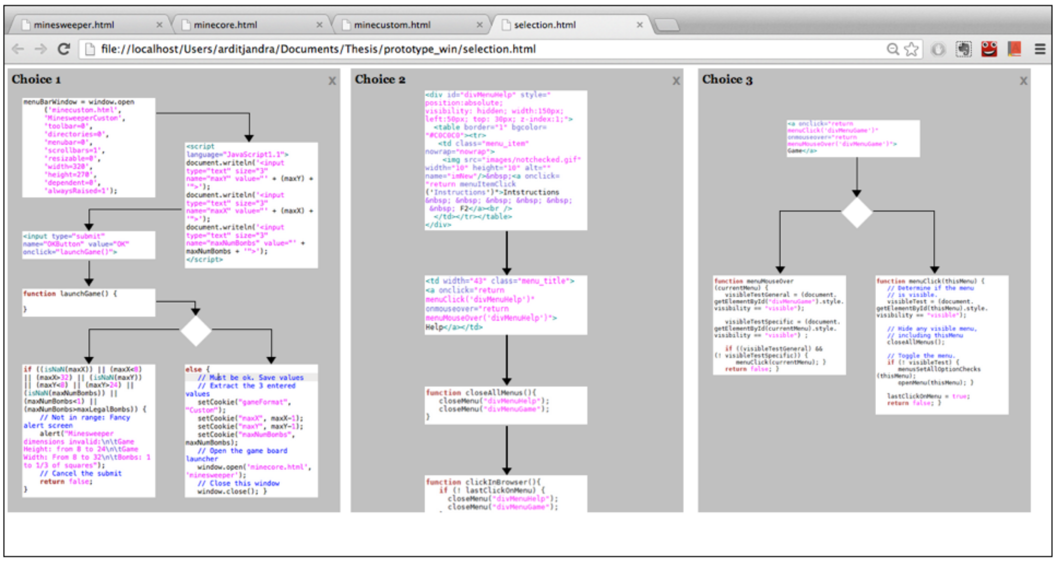
Users prepare for the study by playing around with the simulated environment for a short while.

Once users are comfortable enough with the environment, they are presented with several tasks, each of them belongs to a certain category. The first task category is *code comprehension*, and it involves asking the users to describe how the Minesweeper code fulfills various functionalities. Users are given four tasks with different levels of complexity, from a straightforward inquiry about the pre-condition and the result of executing a single function, to questions that require them to contrast two similar functions and identify multiple parts of code segments that are related to a single functionality. It bears repeating that the code that is presented to the users are actually a group of images, so users could not perform textual searches on the code. They are therefore encouraged to use the pre-made compositions to help them finish the tasks, but are otherwise given full freedom to use an alternative approach (e.g. pens and papers).

The first task category is *code comprehension*, which inquires users on how the code is written to achieve various functionalities.

The second task category is *information foraging*, and it involves giving the users a code modification task which can be solved by using an existing composition. Users are to select the most relevant composition from a group of three (to emulate "search results"), and then they are asked to incorporate its content into the existing code. The titles of these compositions are deliberately removed, due to our interest in finding out whether the solution graphs alone are sufficient for users to determine the most relevant composition (fourth hypothesis, see Section 6.1 in p. 91). Once they have chosen one composition out of the three, users are asked

The second task category is *information foraging*, which asks users to select a composition and integrate its content into the existing code.



Choice 1

```
menuBarWindow = window.open
      ('minecuston.html',
      'MinesweeperCustom',
      'toolbar=0',
      'directories=0',
      'menubar=0',
      'scrollbars=1',
      'resizable=0',
      'width=320',
      'height=270',
      'dependent=0',
      'alwaysRaised=1');
```

```
<script
language="JavaScript1.1">
document.writeln('<input
type="text" size="3"
name="maxY" value="' + (maxY) +
'">');
document.writeln('<input
type="text" size="3"
name="maxX" value="' + (maxX) +
'">');
document.writeln('<input
type="text" size="3"
name="maxNumBombs" value="' +
maxNumBombs + '">');
</script>
```

```
<input type="submit"
name="OKButton" value="OK"
onclick="launchGame()">
```

```
function launchGame() {

}
```

```
if ((isNaN(maxX)) || (maxX<8)
|| (maxX>32) || (isNaN(maxY))
|| (maxY<8) || (maxY>24) ||
(isNaN(maxNumBombs)) ||
(maxNumBombs<1) ||
(maxNumBombs>maxLegalBombs)) {
    // Not in range: Fancy
alert screen
    alert("Minesweeper
dimensions invalid:\n\tGame
Height: from 8 to 24\n\tGame
Width: From 8 to 32\n\tBombs: 1
to 1/3 of squares");
    // Cancel the submit
    return false;
}
```

```
else {
   // Must be ok. Save values
   // Extract the 3 entered
values
   setCookie("gameFormat",
"Custom");
   setCookie("maxX", maxX-1);
   setCookie("maxY", maxY-1);
   setCookie("maxNumBombs",
maxNumBombs);
   // Open the game board
launcher
   window.open('minecore.html',
'minesweeper');
   // Close this window
   window.close(); }
```

Choice 2

```
<div id="divMenuHelp" style="
position:absolute;
visibility: hidden; width:150px;
left:50px; top: 30px; z-index:1;">
  <table border="1" bgcolor=
"#C0C0C0"><tr>
   <td class="menu_item"
nowrap="nowrap">
    <img src="images/notchecked.gif"
width="10" height="10" alt=""
name="imNew"/>&nbsp;<a onclick=
"return menuItemClick
('Instructions')">Intstructions
&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;
 &nbsp; F2</a><br />
  </td></tr></table>
</div>
```

```
<td width="43" class="menu_title">
<a onclick="return
menuClick('divMenuHelp')"
onmouseover="return
menuMouseOver('divMenuHelp')">
Help</a></td>
```

```
function closeAllMenus(){
   closeMenu("divMenuHelp");
   closeMenu("divMenuGame");
}
```

```
function clickInBrowser(){
   if (! lastClickOnMenu) {
     closeMenu("divMenuHelp");
     closeMenu("divMenuGame");
```

Choice 3

```
<a onclick="return
menuClick('divMenuGame')"
onmouseover="return
menuMouseOver('divMenuGame')">
Game</a>
```

```
function menuMouseOver
(currentMenu) {
   visibleTestGeneral = (document.
getElementById("divMenuGame").style.
visibility == "visible");

   visibleTestSpecific = (document.
getElementById(currentMenu).style.
visibility == "visible") ;

   if ((visibleTestGeneral) &&
(! visibleTestSpecific)) {
      menuClick(currentMenu); }
   return false; }
```

```
function menuClick(thisMenu) {
   // Determine if the menu
   // is visible.
   visibleTest = (document.
getElementById(thisMenu).style.
visibility == "visible");

   // Hide any visible menu,
   // including thisMenu
   closeAllMenus();

   // Toggle the menu.
   if (! visibleTest) {
      menusSetAllOptionChecks
(thisMenu);
      openMenu(thisMenu); }

   lastClickOnMenu = true;
   return false; }
```

Figure 6.4: In the second phase, users are given a choice of three compositions, and they are asked to select one to fulfill the given task.

to incorporate the revelant graph elements into the existing code. To accommodate this task, we give users access to a code editor that contains all the four files that make up the Minesweeper project. For the sake of simplicity, users are only asked to find the exact locations in the code to which they would "drag" the relevant elements, but they are also allowed to go the distance and actually copy the contents of the selected composition and alter them as necessary. To emulate a typical working session, and to test our fifth hypothesis (see p. 91), users are given more leeway than in the previous category; they can use the various pre-made compositions, as well as perform textual searches to finish the task at hand.

The final task category deals with the creation of a composition.

The final task category deals with the creation of a composition, specifically its solution graph. As we have mentioned, due to technical limitations, we do not provide users with the ability to drag and drop existing code segments into the solution canvas. Instead, users are presented with a group of graph elements on a single page (as if they have just dragged these elements into the canvas themselves), and then asked to construct a graph using these elements. Our main purpose for including this category despite the limited interaction is to gain insight on how varied the resulting graphs would be if users are given full flexibility in building their solution graphs (third hypothesis, see Section 6.1 in p. 91). Furthermore, users are also given the choice to simply sketch their prefered structure on a piece of paper.

Users are given a questionnaire in order to acquire their general sentiments of our design.

Finally, to cap off the sessions, users are given a questionnaire that focuses on the general ideas of our design, as well as the particulars of the user interface and its interaction styles. Naturally, they are also welcome to give their feedbacks on other topics not included in the questionnaire. The findings we gathered from this study are then processed to help us determine the future direction of our design.

6.4 The Results of The Studies

We treat the questionnaire as a *Likert-type* data, and we employ non-parametric procedures to analyze it.

The questionnaire consists of a group of statements, each addresses a particular design detail of Code Mixer. Users are then presented with five different choices that denote their level of agreement with the given statements, from "strongly agree" all the way to "strongly disagree". We treat each statement as an individual *Likert Item* and assign the choices with scores ranging from 1 to 5, with 1 denoting the most positive reaction. Since we are interested in seeing the effectiveness of every single design decision on its own, we treat the results as a *Likert-Type* data, as oppoased to *Likert Scale*, which measures the composite of a series of Likert-type items. Accordingly, we employ non-parametric procedures based on *median* (*Mdn*) to determine the central tendency of each item, but we also calculate the average (*M*) score for each item as supplementary information. As for the variability of the data, we measure the *frequencies* of each choices for each Likert item. The full results of both analyses can be seen in Appendix B.

Statements	*Mdn*	*M*
I think Code Mixer can improve code comprehension	1	1
I think Code Mixer can improve code-related information sharing	1	1.1
I think Code Mixer can improve code-related problem solving	1	1.1

Table 6.1: General Reactions on Code Mixer

6.4.1 General Reaction to Code Mixer

The users react positively to the general idea of Code Mixer in terms of its ability to provide help during maintenance tasks.

To our delight, there is a collective agreement among the study participants that Code Mixer, with its task-oriented compositions and straightforward interaction styles, can indeed help developers with their day-to-day programming activities, particularly maintenance tasks. As a matter of fact, an absolute concensus (*Mdn*: 1, *M*: 1) is reached that Code Mixer would provide help for developers during code comprehension periods (see Table 6.1). Participants are also highly confident (*Mdn*: 1, *M*: 1.1) that Code Mixer could facilicate information sharing, which in turn would improve information foraging. These positive results are achieved despite participants experiencing various degrees of difficulties while finishing the tasks presented in the study, which we attribute to users needing a little more time to familiarize themselves with the code. The user feedbacks have nevertheless given us confidence that the general direction of our design is already on the right track.

6.4.2 Interface Details and Visual Identity

Users reacted positively to the way we depict an association between a code segment and a composition.

The questionnaire is divided into several different categories, the first of which focuses on the various facets of the user interface. To start, users are mostly receptive of our method of depicting an association between a code segment with a composition, which is accomplished by inserting the composition icons in the code editor (*Mdn*: 1 *M*: 1.6, see Table 6.2). They also react positively to the use of

Statements	*Mdn*	*M*
I liked the placements of various icons in the working area as a way to identify existing compositions	1	1.6
I liked the use of a gray box to highlight the part of the working code that is connected to a certain icon	1	1.3
I liked the use of a pop-up window to contain the compositions (as opposed to, for instance, an integrated section in the working area)	3	3.1
I was able to differentiate a certain compositions from the others based on its title and its icon in the working area	2	2
I was able to quickly recognize a certain composition based on the user-generated shapes it contains	2	2.5
I liked the variety of the shapes that the elements of a compositions can have, which is based on the information that they actually contain (as opposed to a set of generic shapes that are provided by the system)	1	1.8
I liked the variety of positions that the elements of a composition can have (as opposed to automatic positioning by the system)	1	1.8

Table 6.2: Reactions on Code Mixer's Interface

gray box to highlight the relevant code segment, which will be displayed when the mouse cursor hovers over the icon (*Mdn*: 1 *M*: 1.3).

Users respond positively to the flexible structure of the solution graph, but have mixed reactions regarding the application window.

Meanwhile, the majority of users (eight people, *Mdn*: 1) indicate a preference for the flexible structure of the solution graphs as opposed to automatic positioning. Similarly, content-based graph elements also receive the majority's positive votes (seven people, *Mdn*: 1), beating the alternative of a combination of generic shapes and labels. The biggest disagreement among users originates from the placement of the application window. With an average score of 3.1 and a median value of 3, users are seemingly evenly divided between the current approach of hosting the composition in a new pop-up window and in an integrated panel in the IDE. Clearly, this is something that we need to ponder about in our future design phases.

Further measures need to be taken when assigning icons to a composition.

While constructing the prototype, we inadvertently assigned the same icon shape to all compositions, with only different colors separating one composition from the others. This would raise an interesting issue during one of the study session. One of the study participants suffers from a partial color-blindness, and thus, struggled to distinguish between some of the icons inserted in the simulated code editor. This case has brought into our attention that some further measures need to be taken when assigning automatic icons for compositions.

Users only use the composition icons to distinguish the different compositions.

Another category included in the questionnaire is the strength of the identity of the compositions. A majority of 8 participants claim to be able to distinguish among different compositions by using their titles and icons. However, further inquiries reveal that most subjects pay attention solely to the icons while ignoring the titles, having become used to treat a window title as a mere design decoration instead of a valuable content. On the one hand, the answers from the questionnaire confirm our first hypothesis (see Section 6.1 in p. 91), but the following finding tells us that it has not been fully satisfied. We would like to equip the compositions with as strong an identity as possible, and therefore we need to come up with a design tweak that highlight the composition title so it can also help users identify a composition. Some users have suggested that to increase the visual identity, the background color of the composition window should be altered according to the color of the icon (not unlike the way iTunes 11 displays album tracks), but we are quite wary of color-related adjustments after the problem we encountered with the aforementioned user with color blindness, and this particular idea has the potential to make the work area too cluttered.

Meanwhile, it has also been mentioned that to improve a composition's visual identity, content-based thumbnail images would act as graph elements, while graph structures should solely be based on users' whims. However, during the study we observed that most users seemed not to notice the variety of structures of the presented solution graphs, which disproves our second hypothesis (see p. 91). Nevertheless, as has been stated above, users did react positively to these features, and some even voiced their agreement that these features can improve a composition's visual identity. This contradiction indicates that users may need some time to become accustomed to the structures of the solution graphs, particularly when the graphs are not self-made.

Users seem not to notice the different structures of the solution graphs, despite reacting positively to our design choices.

6.4.3 Information Clarity and Interaction Style

The last two questionnaire categories, which cover information clarity and interaction styles, find the users in relative agreement with our design decisions, with all statements across the two categories averaging at least a 2, and all but one item has a median value of at least 1.5 (see Table 6.3). In the former category, users mostly manage to understand the information contained by the pre-made compositions, including the intentions of the graph elements. This finding incidentally supports our decision to relegate the pattern-centered properties to the application menu, since their omission from the prototype ultimately did not stop the users from finishing the tasks. Users also find the contents of the graph elements readable, implying acceptance of the text-wrapping technique.

Users have little problem understanding the information contained by the pre-made compositions.

Moreover, users managed to identify the currently active graph element without a hitch, and they were also able to easily locate the associated code segment in the simulated code editor. Nevertheless, some users did propose the suggestion to increase the border size of the active graph element, so as to make it stand out even more. A couple of users were also not aware that placing the mouse cursor over a graph element would bring up their description panel, but when informed of this feature, they all reacted positively. This behavior nevertheless signifies that a design tweak is needed to make the existence of this feature more obvious. Finally, the last questionnaire category indicates that users are mostly satisfied with the interaction

Users are mostly satisfied with the current interaction styles.

Statements	*Mdn*	*M*
I was able to understand the general purpose of each composition presented in the study	1.5	1.6
I was able to understand the specific purpose of each element contained by the compositions presented in the study	1	1.6
I was able to easily access and understand the additional information (i.e., the descriptive texts of the elements) of a composition	1.5	1.8
I was able to identify the parts of the working area which are associated to the elements of the compositions	1	1.4
I was able to easily locate the element of a composition that is currently active	1	1.5
While doing the second task, I found the content of the compositions to be readable	2	2
I liked the idea of activating the application window by clicking an icon in the working area (as opposed to, for instance, having the composition area visible at all times)	1	2
I found it easy to navigate between all the elements of a composition	1	1.3
I found it easy to navigate from the working area to the application window and the other way around	1	1.4

Table 6.3: Reactions on Code Mixer's Information Clarity and Interaction Style

styles that we have proposed, which implies that our decision to make them as conventional as possible has paid off.

6.4.4 Additional Comments

We have mentioned that in the second task category, which was *information foraging*, we presented a set of three composition with no titles to the users, to see if users can find the most relevant information based on their contents alone.

It turned out all users managed to identify the most appropriate composition to solve the given task, and all but one managed to find the appropriate locations in the code editor to which the content should be inserted. This confirms our fourth hypothesis (see Section 6.1 in p. 91), which suggests that properly-made compositions can be easily understood even by those unfamiliar with the code. All users also instinctively utilized Code Mixer to navigate the source code, in combination with textual searches that were allowed in this phase of the study, verifying our fifth hypothesis.

Users managed to locate the most appropriate solution from the content alone.

Meanwhile, in the third task category, we asked users to construct a graph out of a set of elements to examine our third hypothesis regarding the structural variety of the resulting graphs. We did not give the users any other task descriptions, and some users performed this task on the given environment, while the others opted to do it on a piece of paper. Some of the resulting graphs did vary from the others, although half of them took the form of a straight line, either horizonzally or vertically. Therefore, we cannot yet validate the truthfulness of our third hypothesis (see p. 91). We feel that further investigations are required, with a better equipped simulated environment, as well as a properly defined task.

We cannot yet verify our hypothesis regarding the structural variety of users' constructed graphs.

Since the questionnaire only covers the design details of the prototype, it still lacks some of the features that were skipped due to technical limitations. We nevertheless would like to get some opinions on these features, so we had brief discussions with all users about them, even though the users can not get first-hand experience with them yet. As it turned out, a majority of nine users reacted positively to the *zoom-and-pan* interaction style, and vastly preferred it to the alternative of using scrollbars to explore a window with a fixed zoom level. Eight users further agreed that pattern-centered information would increase the value and accessibility of a composition, while the other two felt that this feature may be superfluous.

The majority of users reacted positively to the zoom-and-pan interaction styles and the pattern-centered information.

Moreover, all users reacted positively to the navigation style of dragging and dropping contents between the application window and the code editor, and they had no

All users reacted positively to the drap-and-drop interaction style and the automatic assignment of icons.

problem with entering the composition title manually, although some did express that a default title may have been "nice." Finally, all users also gave positive reactions to our current approach of assigning automatic icon to a composition, while giving them the flexibility to change this icon at any other time.

One promising idea is to give the users the ability to view all compositions that are contained by a single code project.

Outside of the questionnaire, some users also offered some ideas that they believed could improve the overall system. One particularly interesting concept was presented by two users in two separate occasions, and the idea roughly revolves around providing the users with a way to view all the compositions that are contained by a single code project. They argued that this feature would further improve the accessibility of compositions, especially for the cases in which users forget the exact code segments that have been associated with a composition, something that we are in agreement with. When this idea was presented to the other users in the following study sessions, they all reacted positively to this feature.

Some users asked for more information when the mouse cursor is placed over the composition icon.

Another user also raised the valid notion that in many cases, opening the application window to display the composition may not be necessary. For instance, sometimes users may only need a reminder of the composition title of a certain composition icon, and therefore when user places the mouse cursor over the icon, it may be a good idea to present the user with this information. Ultimately, these last two ideas, along with the various findings from the user study, are taken into account when we make the final adjustments to our design, which will be elaborated in the next section.

6.5 Final Design Adjustments

Although no change is required for the essence our design, some design details still need to be further adjusted.

The user studies demonstrate that users react very positively towards the essence of our design, so no fundamental twist is needed in this area. Code Mixer would remain as we intended it to be: providing users with a task-oriented outlook of code through lightweight creation, distribution and reusing of compositions. However, the studies did showcase some design details that require some further adjustments in order to better achieve our design goals.

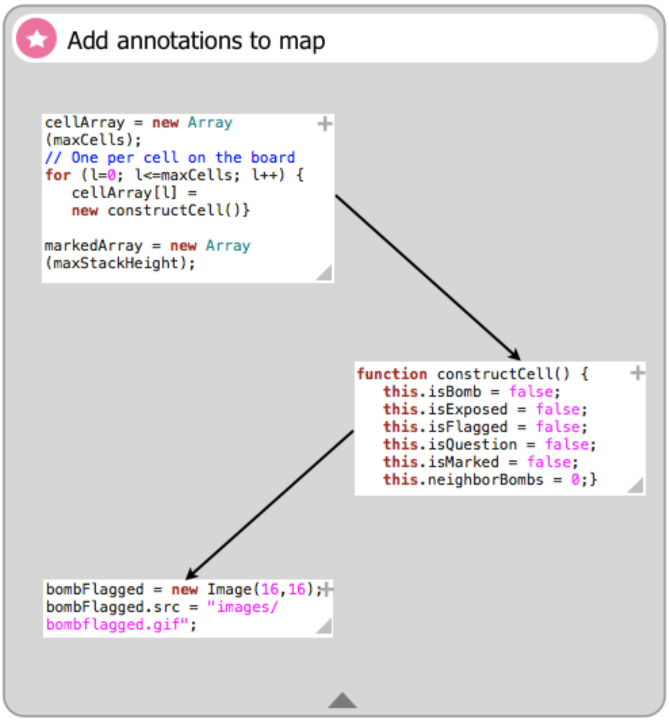
Despite some concerns from the users, we maintain our current approach of hosting the application in a pop-up window.

The very first thing that we would address is the concept that has generated the biggest disagreement among the users: the placement of the application container. Exactly half of the users are in favor of our current approach, while the other half are more partial to having Code Mixer as an integrated part of the IDE. The proponents of the latter argue that an integrated panel would make the overall experience more natural. They claim that it can save users some time, since they would be able to activate the panel just one time, and afterwards they no longer have to click on the composition icons in the IDE. These are valid points, but we still maintain that our current approach offers more flexibility, both for the users and the eventual producers of the system. Hosting the application in a separate window means that from the get go, Code Mixer would have the versatility to work with many different types of IDE, since there is no need to tailor the interface to suit the interface convention of each IDE. Morever, a pop-up window would give the users the possibility to modify the position and size of the solution canvas, and furthermore, they can simply let the application stay hidden in the background when it is not needed. Ultimately, our final decision is to maintain our current approach, despite some concerns from the users.

Users only focuses on one of the three elements that we intended to improve the visual identity of a composition.

Another concept that did not test as well as our expectation is the visual identity of the compositions. As it turned out, most users only pay attention to the composition icons for identification, which is only one of the three elements that are supposed to strengthen a composition's identity, the other two being the composition title and the structure of the solution graph. Our intention is for a composition to bring the users an instant reminder of its content, and therefore a strong visual identity is crucial. We hypothesize that with more exposure time to the solution graphs, users would gradually learn to distinguish their different structures, although further study is required to really confirm this assumption.

The one element that we can fix right now is the apparent "invisibility" of the composition title, which may be caused



```
cellArray = new Array
(maxCells);
// One per cell on the board
for (l=0; l<=maxCells; l++) {
   cellArray[l] =
   new constructCell()}

markedArray = new Array
(maxStackHeight);
```

```
function constructCell() {
   this.isBomb = false;
   this.isExposed = false;
   this.isFlagged = false;
   this.isQuestion = false;
   this.isMarked = false;
   this.neighborBombs = 0;}
```

```
bombFlagged = new Image(16,16);
bombFlagged.src = "images/
bombflagged.gif";
```

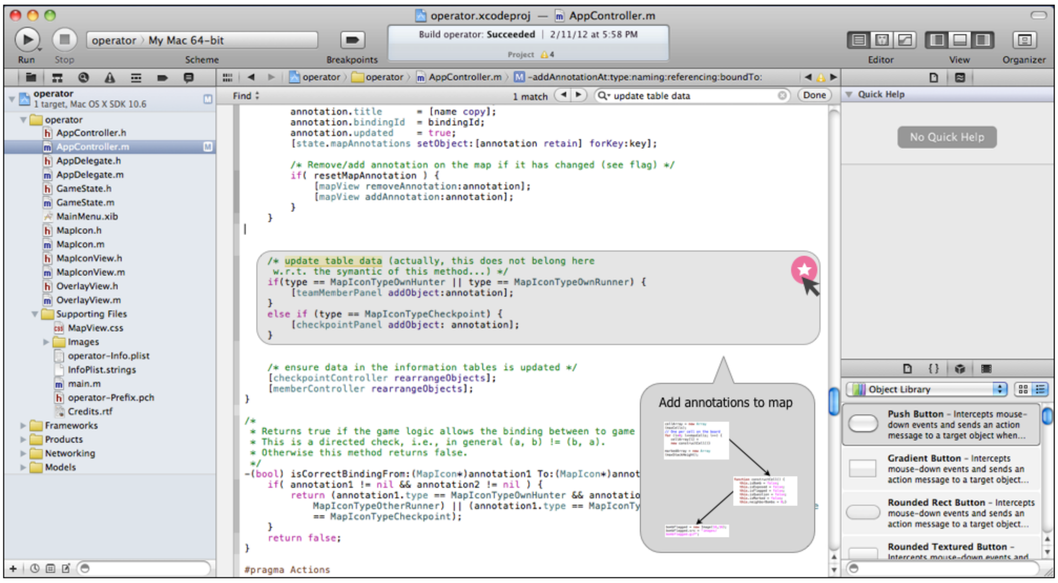
Figure 6.5: The new interface adds new elements in the graph to signify available operations. It also contains a more conspicuous space to host the title.

The composition title is given its own segment in the application window, with a contrasting background from the solution canvas.

by the tendency to treat the title as a mere extension of the window decoration. We attribute this problem to the lack of contrast between the canvas background and the title, which is exacerbated by the fact that the title is sharing the same area with the solution graph, and simply loses the fight for attention against the more eye-catching graph. The solution is pretty straightforward: the composition title no longer floats on top of the solution canvas, but occipies its own segment in the application window. It is further given a contrasting background color in order to emphasize its isolation from the canvas (see Figure 6.5).

Some users took a long time to discover that the are supposed to click on the graph elements.

One problem that caught us off guard during the study was the two occurences in which users failed to notice on their own that placing the mouse cursor over a graph elements would bring out its description panel. It has been established that our intention is for the users to click on the var-



operator.xcodeproj — AppController.m
Build operator: Succeeded | 2/11/12 at 5:58 PM
Run
Stop
operator › My Mac 64-bit
Scheme
Breakpoints
Editor
View
Organizer
operator › operator › AppController.m › -addAnnotationAt:type:naming:referencing:boundTo:
Find
1 match
update table data
Done
Quick Help
No Quick Help
Object Library
Add annotations to map

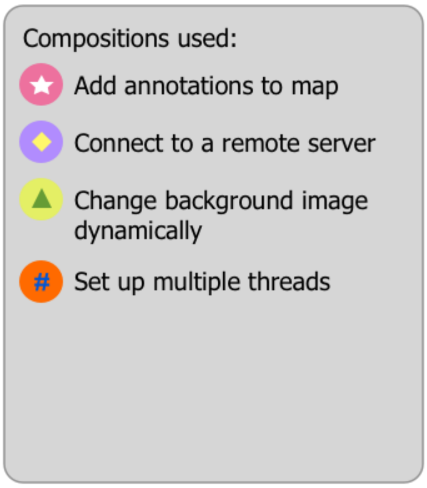
```
        annotation.title      = [name copy];
        annotation.bindingId  = bindingId;
        annotation.updated    = true;
        [state.mapAnnotations setObject:[annotation retain] forKey:key];

        /* Remove/add annotation on the map if it has changed (see flag) */
        if( resetMapAnnotation ) {
            [mapView removeAnnotation:annotation];
            [mapView addAnnotation:annotation];
        }
    }

    /* update table data (actually, this does not belong here
     w.r.t. the symantic of this method...) */
    if(type == MapIconTypeOwnHunter || type == MapIconTypeOwnRunner) {
        [teamMemberPanel addObject:annotation];
    }
    else if (type == MapIconTypeCheckpoint) {
        [checkpointPanel addObject: annotation];
    }

    /* ensure data in the information tables is updated */
    [checkpointController rearrangeObjects];
    [memberController rearrangeObjects];
}

/*
 * Returns true if the game logic allows the binding between to game
 * This is a directed check, i.e., in general (a, b) != (b, a).
 * Otherwise this method returns false.
 */
-(bool) isCorrectBindingFrom:(MapIcon*)annotation1 To:(MapIcon*)annot
    if( annotation1 != nil && annotation2 != nil ) {
        return (annotation1.type == MapIconTypeOwnHunter && annotatio
            MapIconTypeOtherRunner) || (annotation1.type == MapIconTy
            == MapIconTypeCheckpoint);
    }
    return false;
}

#pragma Actions
```

Figure 6.6: Placing the mouse cursor over the composition icon in the IDE displays the composition title and a thumbnail image of the solution graph.

ious graph elements to navigate the code efficiently, and since clicking inevitably requires mouse hovers, it would simultaneously activate the hidden panel. However, two users in particular took a long time before they started to navigate the given code using the composition window, probably due to the lack of clear signifiers that users are supposed to do anything with the graph elements. To remedy this, we decided to add two small design elements that should catch the users' attention.

The first new element is an arrow heads at the bottom right corner of a graph element, which indicates "further information available," and coversely, a lack of this symbol implies that no descriptive text is available for a certain element. Users then no longer have to perform mouse hover on all graph elements just to verify the existence of a description, the way the previous design would require them to. The second design element that has been added to the graph element serves a similar purpose, only it signifies the existence of a set of possible operations that could also be accessed through right click on the mouse. This set of operations has been omitted from the prototype, but they are actually quite important, since they are the ones that al-

Two design elements have been added, one to signify the existence of a descriptive text, and the other to indicate a set of available operations.



Compositions used:

- ★ Add annotations to map
- ◆ Connect to a remote server
- ▲ Change background image dynamically
- # Set up multiple threads

Figure 6.7: The composition library lists all the compositions that have been used by a code project. It is contained in the same application window that hosts all the other Code Mixer components.

low user to add or modify the descriptive text of a graph element. Accordingly, at the top right corner of each element, a small "+" (plus) sign has been added, to signify that "there is more to do here" (see Figure 6.5). Furthermore, despite being a relatively minor tweak, it bears mentioning that we have also decided to take up the user suggestion to increase the size of the border of the selected graph element to make it even more obvious.

Users are presented with more information when the mouse cursor is on a composition icon.

Finally, the last design adjustments involves incorporating two new features, both of which originated from user study participants. The first additional feature provides the users with more information when the mouse cursor is placed on top of the composition icons in the IDE. Where before this action would only highlight the relevant code segment, it now additionally presents the users with the composition title and a thumbnail view of the solution graph (See Figure 6.6). The feature would be particularly beneficial for those cases in which users only need a quick reminder of a certain composition, and it would spare them the requirement to open the application window every single time.

The second and final feature to be added into the design is a local *composition library* that supplies the users with a list of all the compositions that are contained by a single code project. To maintain consistency, the library is displayed in the same application window, and it can be accessed through a keyboard shortcut or selecting the appropriate application menu (See Figure 6.7). With this feature, we give users a new method of accessing the compositions, which adds to the previous two methods of clicking the composition icon in the IDE and opening the composision file in their local machine. The library eliminates the need to browse the entirety of a code project to investigate all existing compositions, a task that can easily become cumbersome for very large projects. Ultimately, it provides a much more efficient method to achieve our design goal of providing the users with a task-oriented outlook of code.

A composition library is provided to let users quickly view all the composisions that have been used by a project.

Chapter 7

Summary and future work

In this paper we present a design for Code Mixer, a tool that provides assistance to developers throughout the entirety of the maintenance period. Section 7.1 briefly sums up the work that has been undergone in this research, while in section 7.2 we present some research areas that still require further investigations.

7.1 Summary and Contributions

Developers are presented with many narrow-focused tools which they tend to neglect.

When facing a maintenance task, developers typically spend some time trying to understand code (*code comprehension*) before they perform code modification, which is frequently done by looking for, and subsequently tailoring, existing solutions found on the Web (*information foraging*). Various tools have been proposed to help developers finish their maintenance-related tasks, but most of them have a rather specific focus, dealing with only one of the two stages of the maintenance phase. Developers are consequently required to utilize a number of tools to address different types of problems, despite the finding by various research that they are not really inclined to make use of these tools in the first place.

Code Mixer hosts compositions that are rooted in real world tasks, while providing more comprehensive help for developers.

Accordingly, we set out to devise a design for Code Mixer, a tool that can help solve developers' maintenance-related problems in a more comprehensive manner. We introduce the two user roles, *composers* and *consumers*, as well as the pieces of information that are shared between the two, called the *compositions*. Our approach is to revolve our design around the creation of compositions, which are rooted in real world tasks (or problems), regardless of their types. Three design principles are established for a tool that support this scheme, namely an emphasis on visual representations, lightweight creation and reuse of compositions and equipping the compositions with task-related context. We also propose a few other attributes that would improve our design, which include accessibility, modifiability, recordability, as well as supporting collaborative effort and giving them spatial freedom when building the compositions.

The design period consists of four phases that build incrementally over time.

The design period of Code Mixer is split up into four phases that build incrementally over time. In the first phase, we identify the need to provide a standardized structure for task descriptions, which is fulfilled by borrowing the *design patterns* format. In the second design phase, the idea of a central repository that hosts the compositions is introduced, along with a complementary pair of desktop applications that serve to connect the users and the repository.

Code Mixer is created out of the desire to balance the workload and unify the user experience of the two user roles.

The notion of Code Mixer is finally realized during the third design phase, which arises out of the desire to balance the workload and unify the user experience of the two user roles, so as to encourage more users to actively assume both roles. A single application window is designed to contain all the task-related information of a composition, and users are allowed to determine the structure of their solution graphs. To further increase the visual identity of a composition, its solution graphs are made out of thumbnail images of its actual content. Furthermore, compositions icons are scattered across the code editor and they serve to identify the code segments that are associated with a certain composition. In the end, users can easily create compositions only by dragging and dropping elements between the code editor and the application window, and they can jump into the relevant code segments in the code editor by simply traversing the solution graph. The user interface is ultimately given further tweaks in the last design phase to make it as efficient as possible. This is the reason that in the final version of the design, the entirety of the application window serves as a canvas for the solution graph.

User studies are conducted to gauge user reactions.

To test the validity of our design, we constructed a web browser-based simulated environment which contained most of the features that we had designed. We then conducted qualitative user studies with 10 users to acquire their overall sentiments and ideas on our proposed design. Study participants unanimously agreed that Code Mixer could be helpful for both code comprehension and information foraging, although some of the user interface details did receive a more mixed reaction.

The results of the user studies are used to incorporate final tweaks into the design and two new features.

Based on the findings from the user studies, we then performed some final tweaks to the user interface design, on top of adding two new features. The first is an enhanced overview of a composition's content when users point the mouse cursor over its icon in the code editor, which will save the users some time when all they need is a quick reminder of a composition's content. The second additional feature is the Composition Library, which provides the users with a quick glimpse of all the compositions that are used by a code project. This feature would grant the users easy access to all available compositions in a code project without having to traverse through all the source code files to track the existence of composition icons.

Code Mixer stands out because it bridges the gap between code comprehension and information foraging.

Ultimately, what separates Code Mixer from most other help tools would be the way it bridges the gap between code comprehension and information foraging. It helps information foraging by supplying a standardized structure for task descriptions and ensuring easy creation, distribution and reuse of compositions. Meanwhile, it improves code comprehension by providing a task-oriented outlook into source code, as well as efficient navigating method that lets users focus only on the task at hand. Moreover, since our proposed design of Code Mixer has been approved by users and manages to fulfill all of our established design principles and properties, we are confident that it could eventually accomplish our goal of helping developers throughout the entirety of the maintenance period.

7.2 Future Work

Code Mixer has managed to fulfill most of our design goals and is well receiced by the users. Nevertheless, there are still some areas that need to be addressed and some research questions that require further investigations.

Implementation

XML can be used to build a composition, as it has a flexible structure and can support information exhange across the Web.

Given that the purpose of this thesis work is to formulate a *design* for Code Mixer, we have not really covered the technical aspects of implementing the tool. One of the most fascinating parts may be the method used to actualize the compositions, as our design primarily revolves around the creation and distribution of compositions. One existing approach that potentially can be borrowed to implement Code Mixer is that of *Codelets* [Oney and Brandt, 2012], where *Extensible Markup Language* (XML) is used to contain the metadata of a piece of Codelet. Similarly, one can utilize XML to segmentize a composition by assigning a certain tag for each of the pattern-related information. XML is already widely used to exhange information across the Web, which means it can support the distribution of compositions that is very essential to our design scheme. Additionally, XML has the benefit of being flexible, and therefore an XML-based establishment is always ready to adapt to future changes in information structure.

Additional Feature

One possible extension is to allow multiple users to work on a single composition together from multiple machines.

We have mentioned before that the current version of Code Mixer can only support collaborative creation and modification of compositions in a somewhat limited manner. Although multiple persons can work together on a single composition, they still have to do it in a single machine. The alternative is for each member of the collaborative group getting their own turn to modify a composition using their own machine, before sending it to the other colleagues. This method is clearly not very efficient, and it may even result in multiple versions of compositions that are hard to unify. Accordingly, one possible new feature that can be added to the system is the ability to collaboratively work on a single composition from multiple machines. This feature would ensure that Code Mixer better satisfies the *collaboration-friendly* property that we established in the initial design phase.

Further Evaluations

Code Mixer still lacks a meticulous method to search for compositions and organize the search results.

One research area that has not been extensively covered in this thesis work is an efficient method to search for a composition. Thus far, we rely solely on the design pattern formats to help users choose the most relevant composition that would suit their problems from a list of search results. However, further studies are required to verify whether our selected approach is indeed effective. There are also more research areas that can still be explored, which include devising the best technique to retrieve compositions from the repository based on user-entered search terms, along with the most effective way to sort and display the search results.

More research is required to accomplish automatic icon assignment.

Another potential research area touches on the automatic icon assignments for a composition once it is download to a code project. An algorithm that can perform such task efficiently still needs to be devised. It bears reiterating that based on the findings from the user studies, the composition icons should not be too dependant on colors, which implies that more attention should be given to the shapes of the icons themselves or the graphic that it contains, if any. We would refer interested readers to the work of Lewis et al. [2004] which proposes automatic drawing of various desktop icons to increase the visual identities of documents, since it covers roughly the same area of interest.

It has yet to be determined that spatial memory can gradually build over time.

Finally, observations during the user studies pose one interesting question regarding the effectiveness of the freeform structures of the solution graphs. We have mentioned that most users prefered to be given spatial freedom when building the graph, and we intented to take advantage of each user's personal touch to improve the visual identity of a composition. However, during the studies we noticed that most users did not pay attention to the actual forms of the presented graph structures, a tendency that we attributed at the time to the users needing more time to familiarize themselves with the compositions. Consequently, further research is required to determine if spatial memory would indeed build gradually over time, as well as to investigate the methods that can be used to speed up this process.

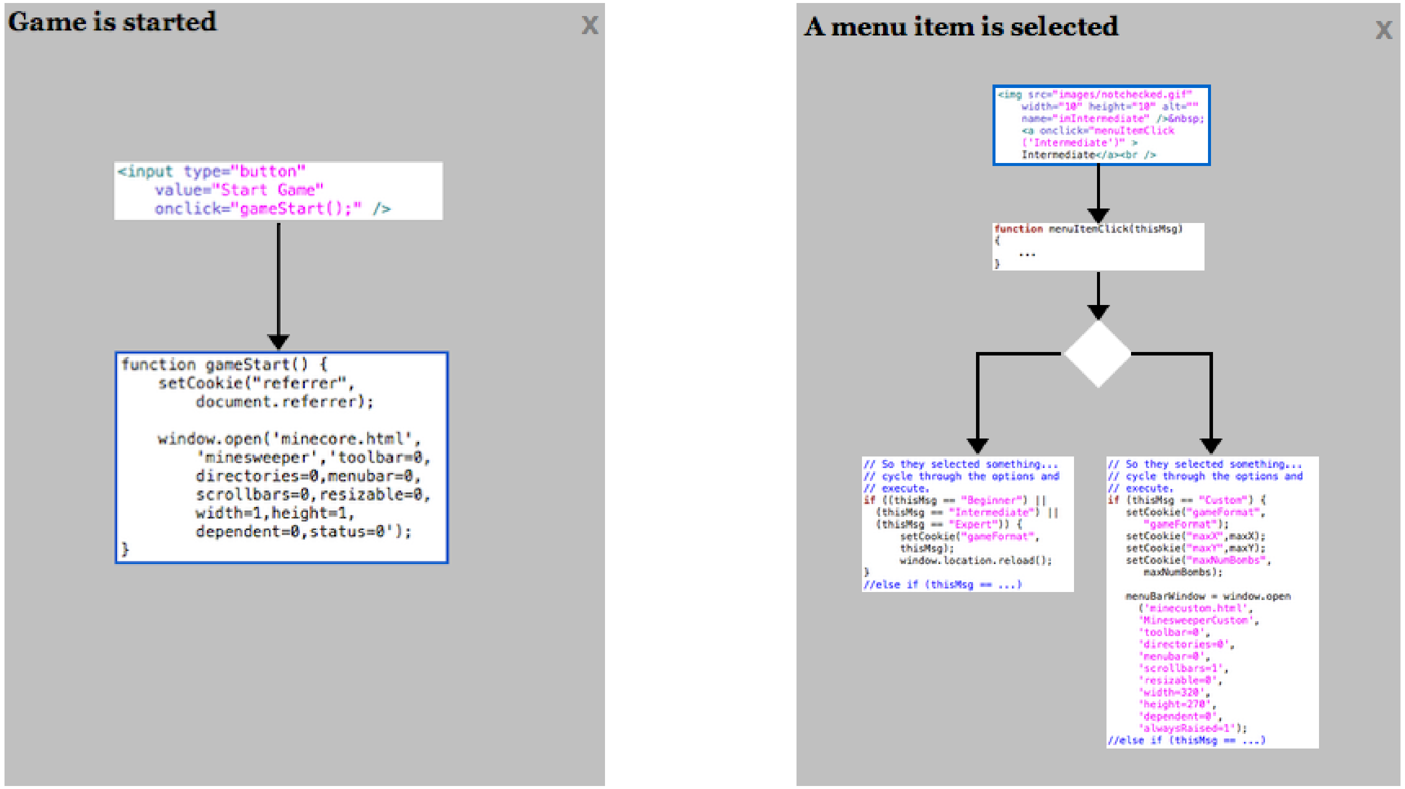
Appendix A

Detailed Views of the Prototype

A.1 Code Comprehension

In the first phase of the study, we present the users with five pre-made compositions that they can utilize to solve the given code-comprehension tasks. They are the following:

- Composition #1: Game is started
  User clicks on a button on the welcome page, which would evoke a function that opens the game in a new window.
- Composition #2: A menu item is selected
  User selects a game level from the menu. When a normal level is selected, the game simply reloads. When a custom level is selected, user is presented a new window to input the desired game properties.
- Composition #3: Custom level is selected
  User inputs game width, height, and number of bombs. The system then verifies if user inputs are valid or not. If they are, the game reloads with the new values, otherwise, an error message is displayed.
- Composition #4: When a bomb is clicked
  If a cell that contains a bomb is clicked, the game changes all cells into the appropriate images and stops all game functions.
- Composition #5: Initialize a game
  Setting up the game, including building the game board and placing bombs at random locations, based on the game level.



Game is started

```
<input type="button"
    value="Start Game"
    onclick="gameStart();" />
```

```
function gameStart() {
    setCookie("referrer",
        document.referrer);

    window.open('minecore.html',
        'minesweeper','toolbar=0,
        directories=0,menubar=0,
        scrollbars=0,resizable=0,
        width=1,height=1,
        dependent=0,status=0');
}
```

A menu item is selected

```
<img src="images/notchecked.gif"
    width="10" height="10" alt=""
    name="imIntermediate" />&nbsp;
    <a onclick="menuItemClick
    ('Intermediate')" >
    Intermediate</a><br />
```

```
function menuItemClick(thisMsg)
{
    ...
}
```

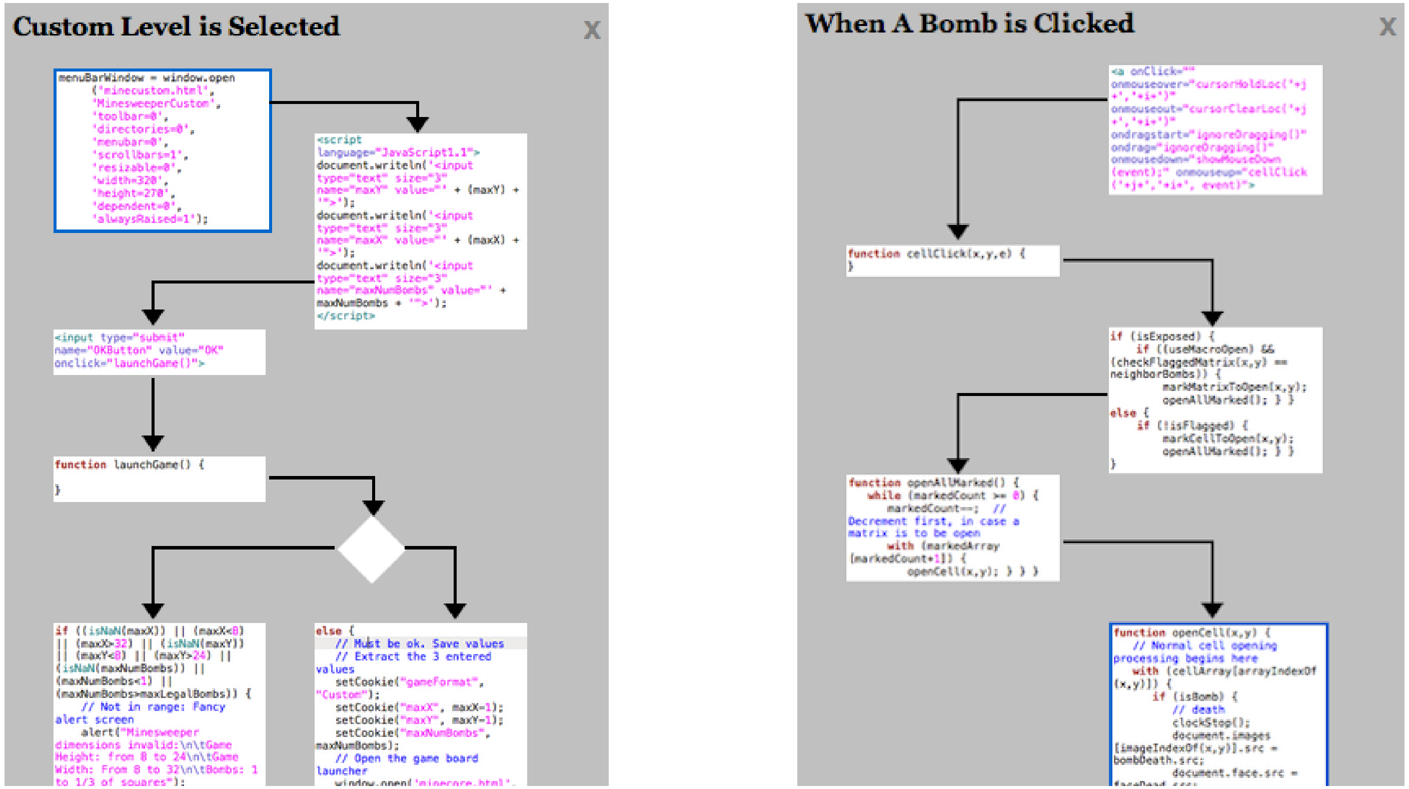
```
// So they selected something...
// cycle through the options and
// execute.
if ((thisMsg == "Beginner") ||
  (thisMsg == "Intermediate") ||
  (thisMsg == "Expert")) {
      setCookie("gameFormat",
      thisMsg);
      window.location.reload();
}
//else if (thisMsg == ...)
```

```
// So they selected something...
// cycle through the options and
// execute.
if (thisMsg == "Custom") {
   setCookie("gameFormat",
      "gameFormat");
   setCookie("maxX",maxX);
   setCookie("maxY",maxY);
   setCookie("maxNumBombs",
      maxNumBombs);

   menuBarWindow = window.open
     ('minecustom.html',
     'MinesweeperCustom',
     'toolbar=0',
     'directories=0',
     'menubar=0',
     'scrollbars=1',
     'resizable=0',
     'width=320',
     'height=270',
     'dependent=0',
     'alwaysRaised=1');
//else if (thisMsg == ...)
```

(a) Composition #1

(b) Composition #2

Figure A.1: The first two compositions in the first task.



Custom Level is Selected

```
menuBarWindow = window.open
      ('minecustom.html',
      'MinesweeperCustom',
      'toolbar=0',
      'directories=0',
      'menubar=0',
      'scrollbars=1',
      'resizable=0',
      'width=320',
      'height=270',
      'dependent=0',
      'alwaysRaised=1');
```

```
<script
language="JavaScript1.1">
document.writeln('<input
type="text" size="3"
name="maxY" value="' + (maxY) +
'">');
document.writeln('<input
type="text" size="3"
name="maxX" value="' + (maxX) +
'">');
document.writeln('<input
type="text" size="3"
name="maxNumBombs" value="' +
maxNumBombs + '">');
</script>
```

```
<input type="submit"
name="OKButton" value="OK"
onclick="launchGame()">
```

```
function launchGame() {

}
```

```
if ((isNaN(maxX)) || (maxX<8)
|| (maxX>32) || (isNaN(maxY))
|| (maxY<8) || (maxY>24) ||
(isNaN(maxNumBombs)) ||
(maxNumBombs<1) ||
(maxNumBombs>maxLegalBombs)) {
    // Not in range: Fancy
alert screen
    alert("Minesweeper
dimensions invalid:\n\tGame
Height: from 8 to 24\n\tGame
Width: From 8 to 32\n\tBombs: 1
to 1/3 of squares");
```

```
else {
   // Must be ok. Save values
   // Extract the 3 entered
values
   setCookie("gameFormat",
"Custom");
   setCookie("maxX", maxX-1);
   setCookie("maxY", maxY-1);
   setCookie("maxNumBombs",
maxNumBombs);
   // Open the game board
launcher
   window.open('minecore.html',
```

(a) Composition #3

When A Bomb is Clicked

```
<a onClick=""
onmouseover="cursorHoldLoc('+j
+','+i+')"
onmouseout="cursorClearLoc('+j
+','+i+')"
ondragstart="ignoreDragging()"
ondrag="ignoreDragging()"
onmousedown="showMouseDown
(event);" onmouseup="cellClick
('+j+','+i+', event)">
```

```
function cellClick(x,y,e) {
}
```

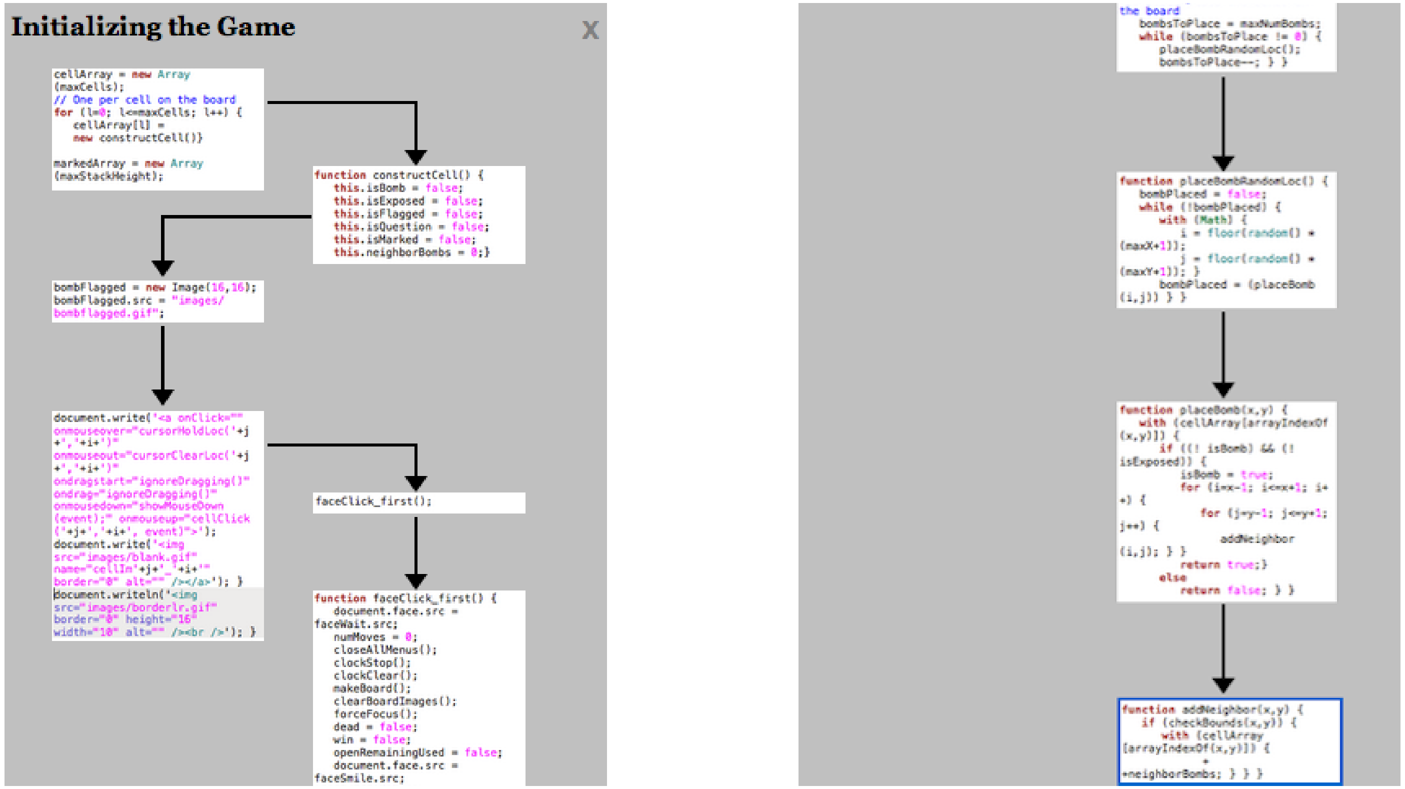
```
if (isExposed) {
    if ((useMacroOpen) &&
(checkFlaggedMatrix(x,y) ==
neighborBombs)) {
        markMatrixToOpen(x,y);
        openAllMarked(); } }
else {
    if (!isFlagged) {
        markCellToOpen(x,y);
        openAllMarked(); } }
}
```

```
function openAllMarked() {
   while (markedCount >= 0) {
      markedCount--;  //
Decrement first, in case a
matrix is to be open
      with (markedArray
[markedCount+1]) {
         openCell(x,y); } } }
```

```
function openCell(x,y) {
   // Normal cell opening
processing begins here
   with (cellArray[arrayIndexOf
(x,y)]) {
      if (isBomb) {
         // death
         clockStop();
         document.images
[imageIndexOf(x,y)].src =
bombDeath.src;
         document.face.src =
faceDead.src;
```

(b) Composition #4

Figure A.2: The second two compositions in the first task



```
cellArray = new Array
(maxCells);
// One per cell on the board
for (l=0; l<=maxCells; l++) {
   cellArray[l] =
   new constructCell()}

markedArray = new Array
(maxStackHeight);
```

```
function constructCell() {
   this.isBomb = false;
   this.isExposed = false;
   this.isFlagged = false;
   this.isQuestion = false;
   this.isMarked = false;
   this.neighborBombs = 0;}
```

```
bombFlagged = new Image(16,16);
bombFlagged.src = "images/
bombflagged.gif";
```

```
document.write('<a onClick=""
onmouseover="cursorHoldLoc('+j
+','+i+')"
onmouseout="cursorClearLoc('+j
+','+i+')"
ondragstart="ignoreDragging()"
ondrag="ignoreDragging()"
onmousedown="showMouseDown
(event);" onmouseup="cellClick
('+j+','+i+', event)">');
document.write('<img
src="images/blank.gif"
name="cellIm'+j+'_'+i+'"
border="0" alt="" /></a>'); }
document.writeln('<img
src="images/borderlr.gif"
border="0" height="16"
width="10" alt="" /><br />'); }
```

```
faceClick_first();
```

```
function faceClick_first() {
   document.face.src =
faceWait.src;
   numMoves = 0;
   closeAllMenus();
   clockStop();
   clockClear();
   makeBoard();
   clearBoardImages();
   forceFocus();
   dead = false;
   win = false;
   openRemainingUsed = false;
   document.face.src =
faceSmile.src;
```

(a) Composition #5 part 1

```
the board
   bombsToPlace = maxNumBombs;
   while (bombsToPlace != 0) {
      placeBombRandomLoc();
      bombsToPlace--; } }
```

```
function placeBombRandomLoc() {
   bombPlaced = false;
   while (!bombPlaced) {
      with (Math) {
         i = floor(random() *
(maxX+1));
         j = floor(random() *
(maxY+1)); }
      bombPlaced = (placeBomb
(i,j)) } }
```

```
function placeBomb(x,y) {
   with (cellArray[arrayIndexOf
(x,y)]) {
      if ((! isBomb) && (!
isExposed)) {
         isBomb = true;
         for (i=x-1; i<=x+1; i+
+) {
            for (j=y-1; j<=y+1;
j++) {
               addNeighbor
(i,j); } }
         return true;}
      else
         return false; } }
```

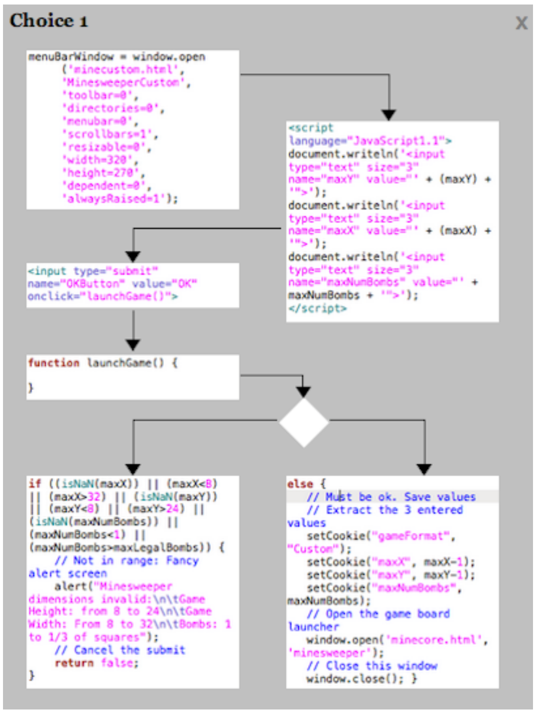
```
function addNeighbor(x,y) {
   if (checkBounds(x,y)) {
      with (cellArray
[arrayIndexOf(x,y)]) {
         +
+neighborBombs; } } }
```

(b) Composition #5 part 2

Figure A.3: The last composition in the first task

A.2 Information Foraging

In the second phase of the study, we ask users to select one of the three give compositions that would best fit the given problem, which is to add a new menu item to the game. The correct answer is the second choice. The first choice is actually Composition #3 from the previous task, and the third choice deals with mouse events in the menu.



```
menuBarWindow = window.open
      ('minecuston.html',
      'MinesweeperCustom',
      'toolbar=0',
      'directories=0',
      'menubar=0',
      'scrollbars=1',
      'resizable=0',
      'width=320',
      'height=270',
      'dependent=0',
      'alwaysRaised=1');
```

```
<script
language="JavaScript1.1">
document.writeln('<input
type="text" size="3"
name="maxY" value="' + (maxY) +
'">');
document.writeln('<input
type="text" size="3"
name="maxX" value="' + (maxX) +
'">');
document.writeln('<input
type="text" size="3"
name="maxNumBombs" value="' +
maxNumBombs + '">');
</script>
```

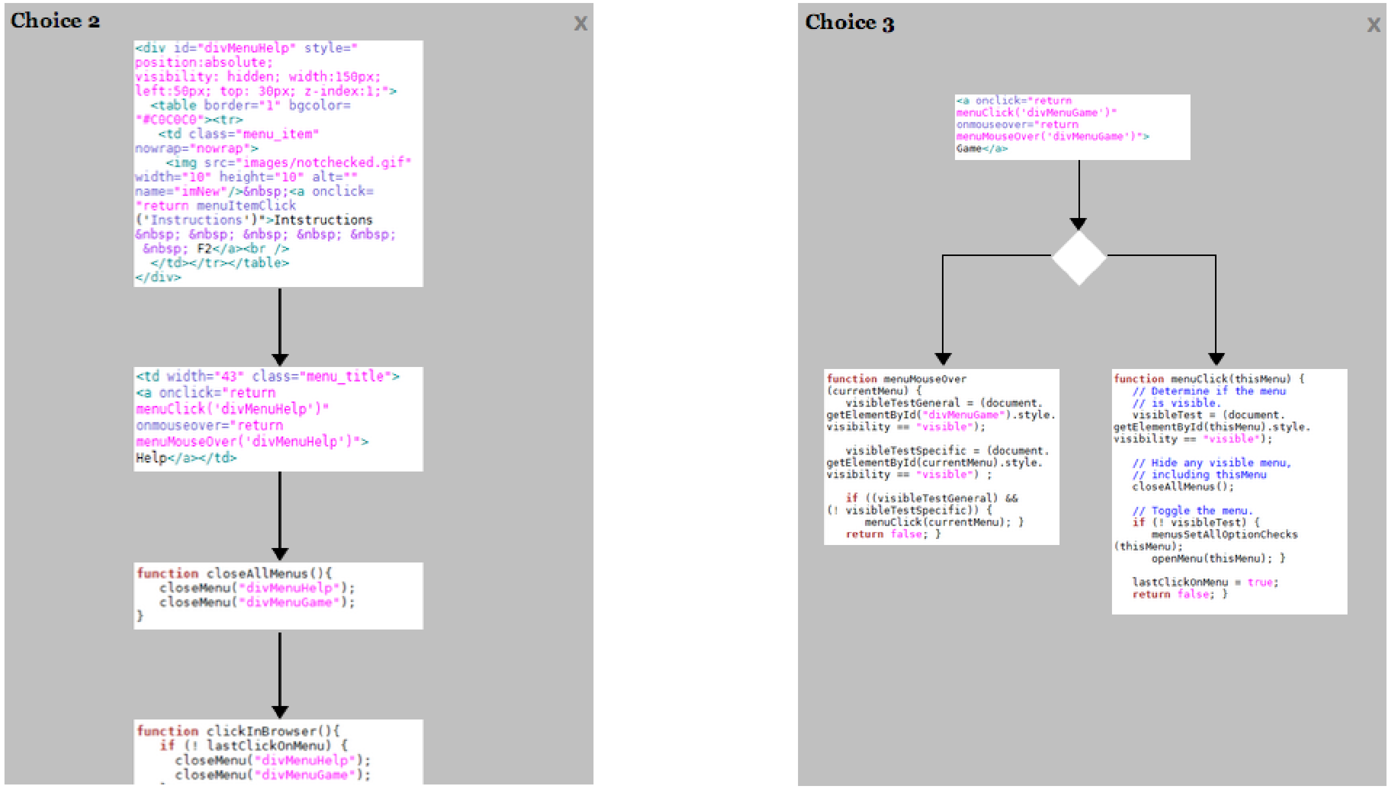
```
<input type="submit"
name="OKButton" value="OK"
onclick="launchGame()">
```

```
function launchGame() {

}
```

```
if ((isNaN(maxX)) || (maxX<8)
|| (maxX>32) || (isNaN(maxY))
|| (maxY<8) || (maxY>24) ||
(isNaN(maxNumBombs)) ||
(maxNumBombs<1) ||
(maxNumBombs>maxLegalBombs)) {
    // Not in range: Fancy
alert screen
    alert("Minesweeper
dimensions invalid:\n\tGame
Height: from 8 to 24\n\tGame
Width: From 8 to 32\n\tBombs: 1
to 1/3 of squares");
    // Cancel the submit
    return false;
}
```

```
else {
   // Must be ok. Save values
   // Extract the 3 entered
values
   setCookie("gameFormat",
"Custom");
   setCookie("maxX", maxX-1);
   setCookie("maxY", maxY-1);
   setCookie("maxNumBombs",
maxNumBombs);
   // Open the game board
launcher
   window.open('minecore.html',
'minesweeper');
   // Close this window
   window.close(); }
```

Figure A.4: The first composition in the second task



```
<div id="divMenuHelp" style="
position:absolute;
visibility: hidden; width:150px;
left:50px; top: 30px; z-index:1;">
  <table border="1" bgcolor=
"#C0C0C0"><tr>
   <td class="menu_item"
nowrap="nowrap">
    <img src="images/notchecked.gif"
width="10" height="10" alt=""
name="imNew"/>&nbsp;<a onclick=
"return menuItemClick
('Instructions')">Intstructions
&nbsp; &nbsp; &nbsp; &nbsp; &nbsp;
 &nbsp; F2</a><br />
  </td></tr></table>
</div>
```

```
<td width="43" class="menu_title">
<a onclick="return
menuClick('divMenuHelp')"
onmouseover="return
menuMouseOver('divMenuHelp')">
Help</a></td>
```

```
function closeAllMenus(){
   closeMenu("divMenuHelp");
   closeMenu("divMenuGame");
}
```

```
function clickInBrowser(){
   if (! lastClickOnMenu) {
     closeMenu("divMenuHelp");
     closeMenu("divMenuGame");
```

(a) The second choice

```
<a onclick="return
menuClick('divMenuGame')"
onmouseover="return
menuMouseOver('divMenuGame')">
Game</a>
```

```
function menuMouseOver
(currentMenu) {
   visibleTestGeneral = (document.
getElementById("divMenuGame").style.
visibility == "visible");

   visibleTestSpecific = (document.
getElementById(currentMenu).style.
visibility == "visible") ;

   if ((visibleTestGeneral) &&
(! visibleTestSpecific)) {
      menuClick(currentMenu); }
   return false; }
```

```
function menuClick(thisMenu) {
   // Determine if the menu
   // is visible.
   visibleTest = (document.
getElementById(thisMenu).style.
visibility == "visible");

   // Hide any visible menu,
   // including thisMenu
   closeAllMenus();

   // Toggle the menu.
   if (! visibleTest) {
      menusSetAllOptionChecks
(thisMenu);
      openMenu(thisMenu); }

   lastClickOnMenu = true;
   return false; }
```

(b) The third choice

Figure A.5: The last two compositions in the second task

A.3 Building a Composition

In the third phase of the study, we ask users to build a solution graph out of the given elements. All elements are draggable, although no arrows are available. Users are also asked to keep in mind the step number contained by each element.

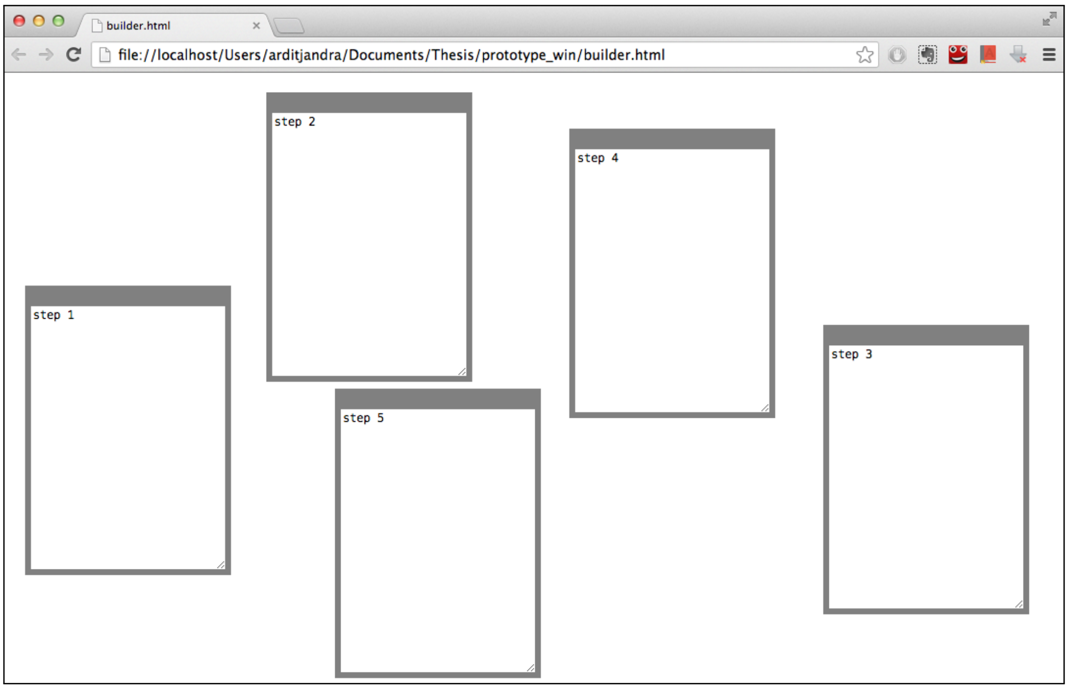

builder.html
file://localhost/Users/arditjandra/Documents/Thesis/prototype_win/builder.html
step 2
step 4
step 1
step 3
step 5

Figure A.6: The simulated environment for the third task

Appendix B

The Results of User Studies

B.1 Background

Num	Statements	*Mdn*	*M*
#1	I am comfortable working with HTML	1	1.2
#2	I am comfortable working with Javascript	1	1.4
#3	I am familiar with the game Minesweeper	1	1.9

Table B.1: Questionnaire result - Background

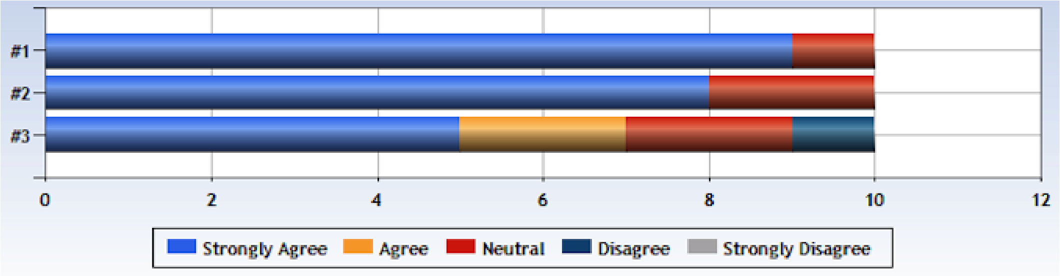


Figure B.1: Questionnaire frequency distribution - Background

B.2 Shapes & Forms

Num	Statements	*Mdn*	*M*
#4	I liked the placements of various icons in the working area as a way to identify existing compositions	1	1.6
#5	I liked the use of a gray box to highlight the part of the working code that is connected to a certain icon	1	1.3
#6	I liked the use of a pop-up window to contain the compositions (as opposed to, for instance, an integrated section in the working area)	3	3.1
#7	I was able to differentiate a certain compositions from the others based on its title and its icon in the working area	2	2
#8	I was able to quickly recognize a certain composition based on the user-generated shapes it contains	2	2.5
#9	I liked the variety of the shapes that the elements of a compositions can have, which is based on the information that they actually contain (as opposed to a set of generic shapes that are provided by the system)	1	1.8
#10	I liked the variety of positions that the elements of a composition can have (as opposed to automatic positioning by the system)	1	1.8

Table B.2: Questionnaire result - Shapes & Forms

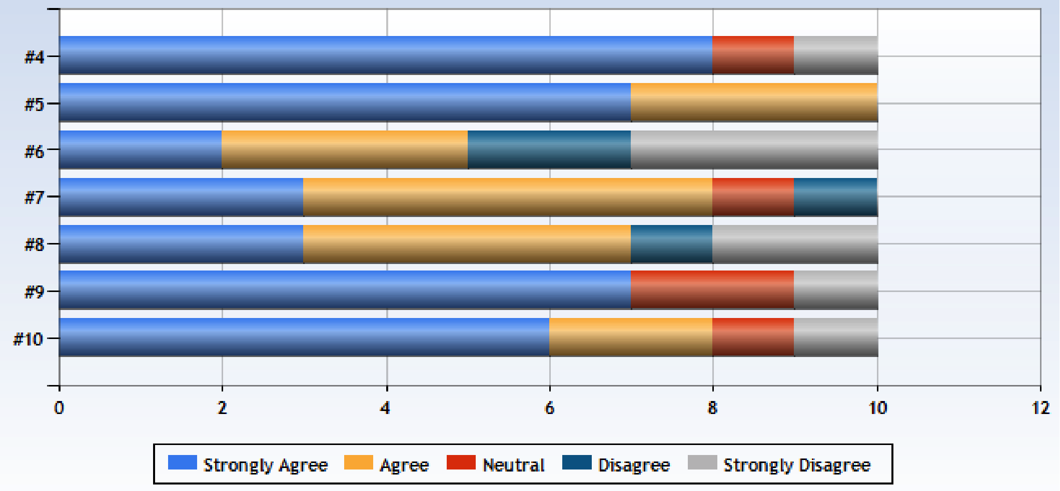


Figure B.2: Questionnaire frequency distribution - Shapes & Forms

B.3 Information Clarity

Num	Statements	*Mdn*	*M*
#11	I was able to understand the general purpose of each composition presented in the study	1.5	1.6
#12	I was able to understand the specific purpose of each element contained by the compositions presented in the study	1	1.6
#13	I was able to easily access and understand the additional information (i.e., the descriptive texts of the elements) of a composition	1.5	1.8
#14	I was able to identify the parts of the working area which are associated to the elements of the compositions	1	1.4
#15	I was able to easily locate the element of a composition that is currently active	1	1.5
#16	While doing the second task, I found the content of the compositions to be readable	2	2

Table B.3: Questionnaire result - Information Clarity

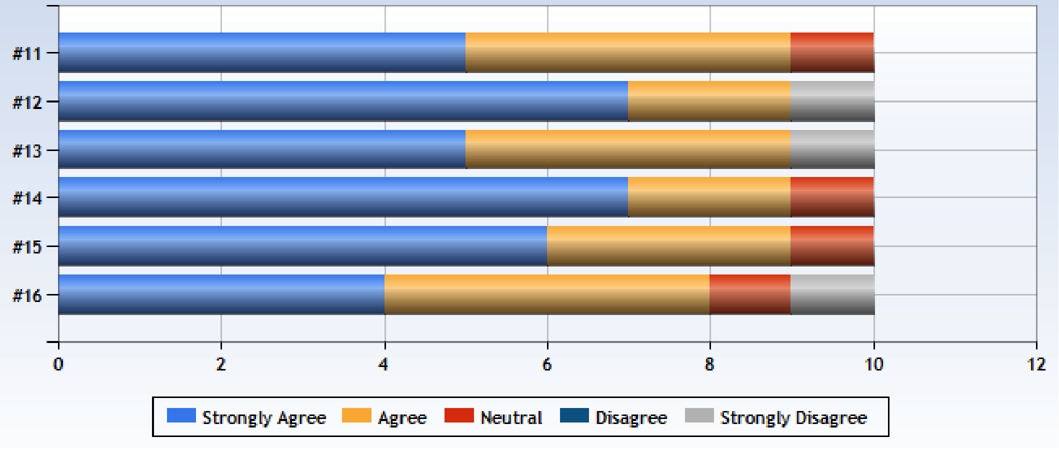


Figure B.3: Questionnaire frequency distribution - Information Clarity

B.4 Interaction Style

Num	Statements	*Mdn*	*M*
#17	I liked the idea of activating the application window by clicking an icon in the working area (as opposed to, for instance, having the composition area visible at all times)	1	2
#18	I liked the idea of activating the gray box highlight with a mouse hover on the icon in the working area (as opposed to, for instance, having the box visible at all times)	1	1.3
#19	I found it easy to navigate between all the elements of a composition	1	1.3
#20	I found it easy to navigate from the working area to the application window and the other way around	1	1.4

Table B.4: Questionnaire result - Interaction Style

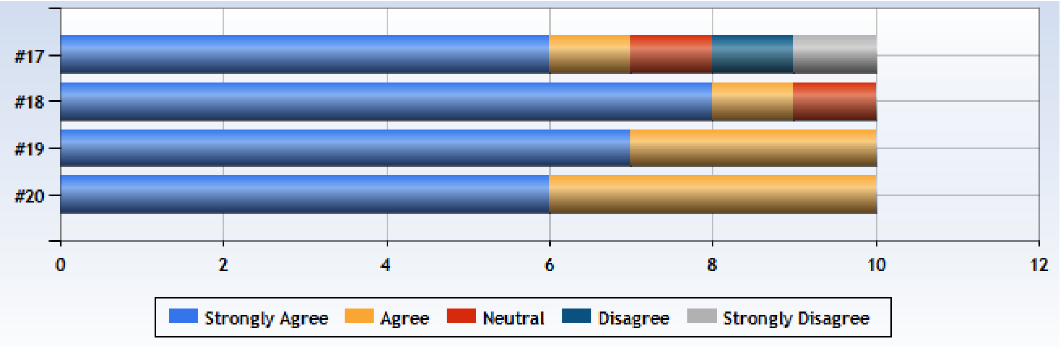


Figure B.4: Questionnaire frequency distribution - Interaction Style

B.5 General Idea

Num	Statements	*Mdn*	*M*
#21	While doing the first task, I found the information that I was looking for easily by using Code Mixer	2	2.1
#22	While doing the second task, I found it easy to select the most appropriate solution to my problem among a group of choices	2	2.2
#23	While doing the second task, I found it easy to integrate elements of a code structure into the working area	2	2.1
#24	While doing the third task, I found it easy to build a composition	2	1.6
#25	I think Code Mixer can improve code comprehension	1	1
#26	I think Code Mixer can improve code-related information sharing	1	1.1
#27	I think Code Mixer can improve code-related problem solving	1	1.1

Table B.5: Questionnaire result - General Idea

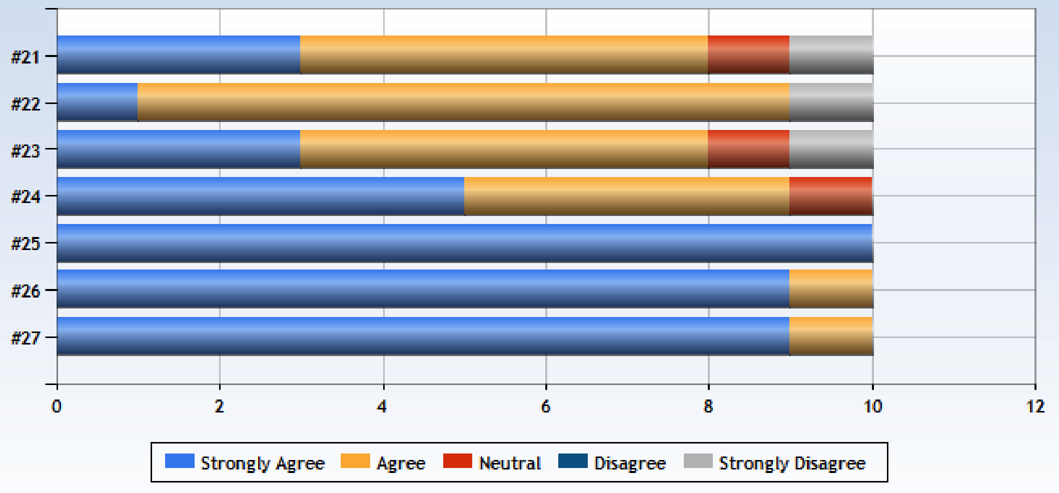


Figure B.5: Questionnaire frequency distribution - General Idea

Bibliography

Christopher Alexander. *The timeless way of building*, volume 1. New York: Oxford University Press, 1979.

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*, volume 2. Oxford University Press, USA, 1977.

Rune Baggetun, Ellen Rusman, and Caterina Poggi. Design patterns for collaborative learning: From practice to theory and back. In *World Conference on Educational Multimedia, Hypermedia and Telecommunications*, volume 2004, pages 2493–2498, 2004.

Jan O Borchers. A pattern approach to interaction design. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, pages 369–378. ACM, 2000.

Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512. ACM, 2010.

Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 513–522. ACM, 2010.

Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini. Ide 2.0: collective intelligence in software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 53–58. ACM, 2010.

Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 557–566. ACM, 2007.

Rylan Cottrell, Robert J Walker, and Jörg Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 214–225. ACM, 2008.

Lotte De Rore, Monique Snoeck, Geert Poels, and Guido Dedene. Software patterns to improve knowledge transfer: an experiment. *Available at SSRN 1376214*, 2009.

Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 207–210. ACM, 2010.

Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven Drucker, and George Robertson. Code thumbnails: Using spatial memory to navigate source code. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 11–18. IEEE, 2006.

Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P Reiss. Debugger canvas: industrial experience with the code bubbles paradigm. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 1064–1073. IEEE Press, 2012.

George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

Andreas Girgensohn and Frank M Shipman. Supporting knowledge acquisition by end users: tools and representations. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990's*, pages 340–348. ACM, 1992.

Paul A Gross, Micah S Herstand, Jordana W Hodges, and Caitlin L Kelleher. A code reuse interface for non-programmer middle school students. In *Proceedings of the 15th international conference on Intelligent user interfaces*, pages 219–228. ACM, 2010.

Björn Hartmann, Mark Dhillon, and Matthew K Chan. Hypersource: Bridging the gap between source and code-related web sites. pages 2207–2210. ACM, 2011.

Reid Holmes, Rylan Cottrell, Robert J Walker, and Jörg Denzinger. The end-to-end use of source code examples: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 555–558. IEEE, 2009.

Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 83–92. IEEE, 2004.

Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent layout for thematic software maps. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 209–218. IEEE, 2008.

Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen Brafman, and Scott R Klemmer. Designing with interactive example galleries. In *Proceedings of the 28th international conference on Human factors in computing systems*, pages 2257–2266. ACM, 2010.

John P Lewis, Ruth Rosenholtz, Nickson Fong, and Ulrich Neumann. Visualids: automatic distinctive icons for desktop interfaces. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 416–423. ACM, 2004.

Chung-Horng Lung, Gerald T Mackulak, and Joseph E Urban. Software reuse and knowledge transfer through analogy and design patterns. In *proceedings of the international conference on software engineering research and practice*, pages 618–624, 2002.

Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.

Fernando Olivero, Michele Lanza, and Mircea Lungu. Gaucho: From integrated development environments to direct manipulation environments. *Proceedings of FlexiTools, 2010*, 2010.

Fernando Olivero, Michele Lanza, Marco D'Ambros, and Romain Robbes. Enabling program comprehension through a visual object-focused development environment. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 127–134. IEEE, 2011.

Stephen Oney and Joel Brandt. Codelets: linking interactive documentation and example code in the editor. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, pages 2697–2706. ACM, 2012.

Peter Pirolli and Stuart Card. Information foraging in information access environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 51–58. ACM Press/Addison-Wesley Publishing Co., 1995.

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, pages 255–265. IEEE Press, 2012.

Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

Vineet Sinha, David Karger, and Rob Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 187–194. IEEE, 2006.

Jamie Starke, Chris Luce, and Jonathan Sillito. Working with search results. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 53–56. IEEE Computer Society, 2009.

M-AD Storey and Hausi A Muller. Manipulating and documenting software structures using shrimp views. In *Software Maintenance, 1995. Proceedings., International Conference on*, pages 275–284. IEEE, 1995.

Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. Shrimp views: an interactive environment for information visualization and navigation. In *CHI'02 extended abstracts on Human factors in computing systems*, pages 520–521. ACM, 2002.

Edward R Tufte and PR Graves-Morris. *The visual display of quantitative information*, volume 2. Graphics press Cheshire, CT, 1983.

Doug Wightman, Zi Ye, Joel Brandt, and Roel Vertegaal. Snipmatch: using source code context to enhance snippet retrieval and parameterization. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 219–228. ACM, 2012.

Index

Typeset September 8, 2013