Aiding Programmers using Lightweight Integrated Code Visualization

Per Ola Kristensson
Department of Engineering
University of Cambridge, UK
pok21@cam.ac.uk

Chung Leung Lam
Computer Laboratory
University of Cambridge, UK
cll48@cantab.net

Abstract

We present a Lightweight Integrated Code Visualization (LICV) tool designed to aid programmers using Integrated Development Environments (IDEs). LICV is implemented as a plug-in for the Eclipse IDE for Java Developers. LICV continuously tracks the active editor in the IDE and visualizes up to 24 code features in a designated non-intrusive view. LICV is designed to facilitate fast understanding of the structure of the code in order to help users carry out routine programming tasks. It enables users to zoom, filter, search, and go back and forth between the code and the visualization via direct manipulation. We evaluated LICV by carrying out two user studies which compared LICV against regular Eclipse in four tasks. We found that LICV significantly reduced participants' completion times by nearly 50% for three out of four tasks. Further, participants significantly preferred using LICV to perform the tasks.

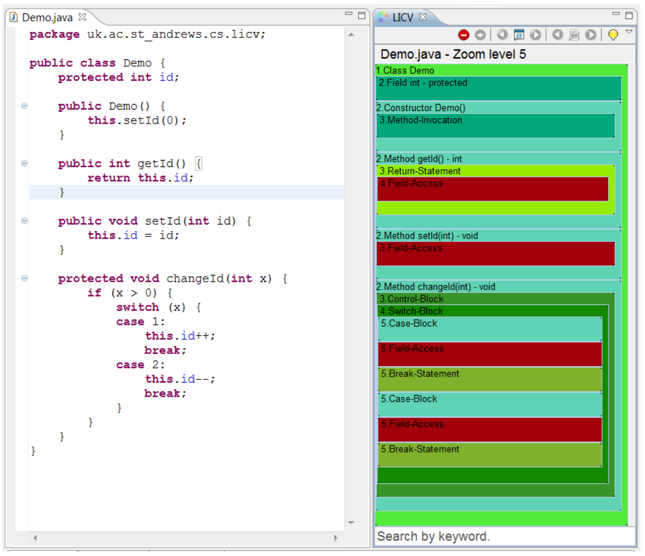
Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Program editors

Keywords Code visualization, integrated development environment, lightweight integrated code visualization

1. Introduction

Programming is a difficult error-prone process and as consequence tremendous research efforts have been devoted to aiding programmers. One approach has been to develop new programming languages and environments, such as Smalltalk. Another example is the code bubbles paradigm [1], which has been implemented in the systems Code Canvas [3] and Debugger Canvas [4].

PLATEAU'15, October 26, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3907-0/15/10...$15.00
http://dx.doi.org/10.1145/2846680.2846683

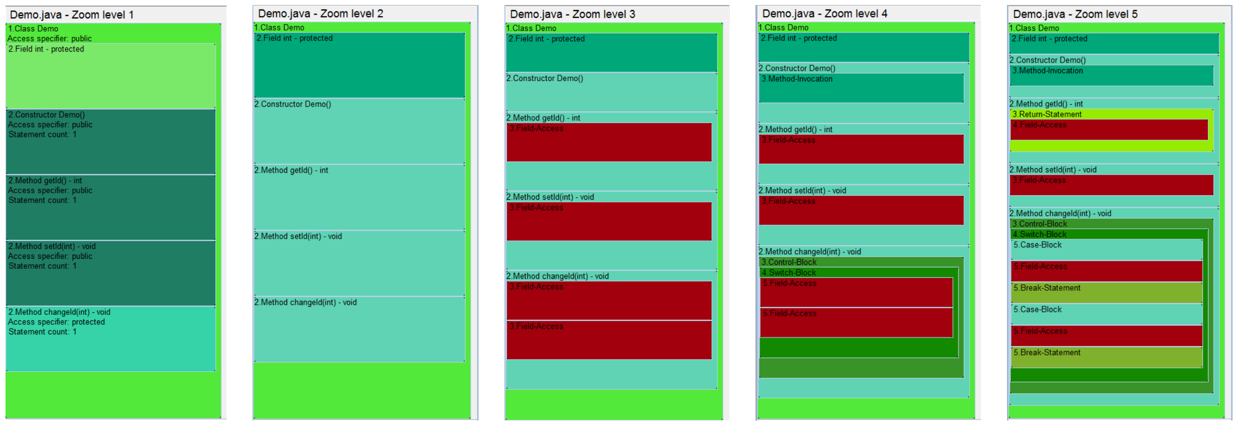


```java
package uk.ac.st_andrews.cs.licv;

public class Demo {
    protected int id;

    public Demo() {
        this.setId(0);
    }

    public int getId() {
        return this.id;
    }

    public void setId(int id) {
        this.id = id;
    }

    protected void changeId(int x) {
        if (x > 0) {
            switch (x) {
            case 1:
                this.id++;
                break;
            case 2:
                this.id--;
                break;
            }
        }
    }
}
```

Figure 1. A Java class in the Eclipse code editor (left) and its visualization in LICV (right). The code window has been resized to the same size as the LICV visualization window for clarity. In actual use the LICV window is considerably smaller than the code window.

A second approach has been to improve existing programming environments that are already widely used. Common approaches include improving programmers' ability to understand and use Application Programming Interfaces (APIs) (e.g. [14, 21]) or improving code autocomplete (e.g. [2, 17]. Some tools tackle particular difficult programming activities, such as debugging, for example the Whyline system [12].

A third approach is to aid programmers via software visualization. Such visualizations extract features (sometimes called software metrics or code metrics) and underlying structures from the code and display them in ways that can aid program comprehension. Various software visualization techniques have been developed, such as Class Blueprints [5], Polymetric Views [13] and fisheyes [7–9].



(a) Overview (b) Zoom 1 (c) Zoom 2 (d) Zoom 3 (e) Zoom 4

Figure 2. Visualization of source code at an overview level and at four different zoom levels.

However, most of these interfaces radically modify a part of the programmer's existing user interface. For example, software visualizations such as Polymetric Views [13] require a large part of the screen estate of the IDE and new autocomplete techniques change programmers' existing behavioral patterns for quickly typing code.

In this paper we explore an alternative *lightweight* approach which does *not* radically change the user interface. Instead, it provides a *complementary* interactive visualization view that enables users to perceive structures in their active code editor at a glance. The closest previous work in this direction is possibly Microprints [19], which visualizes each character in the code as an individual pixel. However, Microprints is a non-interactive visualization. In our work, the underpinning design principle behind our system is to *combine* several useful user interface techniques: hierarchical tree-based code visualization, incremental search, filtering, and bidirectional navigation between the active code editor and the interactive code visualization.

We call this paradigm *Lightweight Integrated Code Visualization (LICV)*. The central hypothesis is that LICV can aid programmers with routine programming tasks, such as for example modifying a single line of code in several similar methods in a class, since it provides the programmer with a non-intrusive complementary structural view of the code.

A LICV is based on the following design principles:

Secondary Interface A LICV *complements* existing code editors and IDEs rather than replaces them. A LICV is designed to aid some subset of tasks rather than serve as a radical complete replacement of existing solutions.

Lightweight A LICV is a calm interface [22]. It achieves this by 1) being unobtrusive; 2) using a structurally stable visualization; 3) preserving screen real-estate for the IDE; and 4) avoiding scrolling by projecting the entire interface and visualization onto visible space.

Visual Information-seeking Mantra A LICV follows "Overview first, zoom and filter, then details-on-demand" [20] by: 1) allowing user-controlled zoom-level (granularity) of the visualization; 2) enabling filtering by elements or by incremental search; and 3) providing direct paths from visualization elements to code.

Smooth Novice to Expert Transition A LICV enables a smooth novice-to-expert transition by being unobtrusive in the IDE. Gradually during occasional use, the programmer becomes increasingly familiar with the LICV visualization and can thereby exploit its advantages without dedicated training sessions.

In order to explore if LICV has a potential to aid programmers in routine programming tasks, we designed a LICV system that complements existing language and IDE paradigms. In the next section we explain how LICV supports programmers and how we implemented it as a plug-in for the Eclipse IDE for Java Developers. Thereafter we report how we evaluated LICV's performance against regular Eclipse. We found that while error rates are nearly identical for both systems, LICV reduces completion times by nearly 50% for three of the four tasks we investigated. We also report how a secondary replication experiment later supported these results.

2. Interactive Visualization of Code Features

Our system is implemented as a plug-in to the Eclipse IDE for Java Developers (version 3.6.1). Users activate LICV by enabling a designed *View* in Eclipse. Figure 1 shows how LICV by default visualizes a small demo class. The elements of the class (called "code features" in this paper; sometimes similar features have been referred to as "software metrics" (e.g. [13])) are shown in a hierarchical tree structure according to the Abstract Syntax Tree (AST) of the class file. Table

1 lists all 24 code features that are currently supported by the system.

Users can zoom in and out in the visualization. This is particularly useful when viewing large class files. Figure 2 shows the Java class in Figure 1 visualized at five different zoom levels.

2.1 Code Features

The system works by extracting 24 features from Java source code files by parsing their Abstract Syntax Trees (ASTs). The ASTs for the Java source code files are generated by the Eclipser Compiler for Java (ECJ). A summary of all 24 code features that are currently supported by the system are shown in Table 1.

Since the system often cannot visualize 24 code features simultaneously we created a ranked list of code features that is used by the system to prioritize among them.

We divided the 24 code features into seven code feature groups: exception handling, local variable declarations, method invocations, field access, iteration constructs, flow control, and unranked code features. Table 1 shows group memberships for all 24 code features.

The ranks were estimated by examining publicly available Java source code files. We first downloaded 5,044 Java source code files with a total of 69,354 methods from five open-source Java projects (Eclipse Java Development Tool, Apache Ant, Apache Derby, SVNKit, and Sweet Home 3D). We then ranked each code feature group based on its code features' frequency of occurrence within each method. This resulted in 69,354 rankings.

We then aggregated these rankings using Kemeny optimal aggregation [10]. We used this particular rank aggregation method because Young and Levenglick [23] have shown that it is the only method that satisfies three attractive criteria for rank aggregation: neutrality, consistency, and the Condorcet criterion. The final aggregated ranks for the code feature groups are shown in Table 1.

Table 1. List of code features with examples.

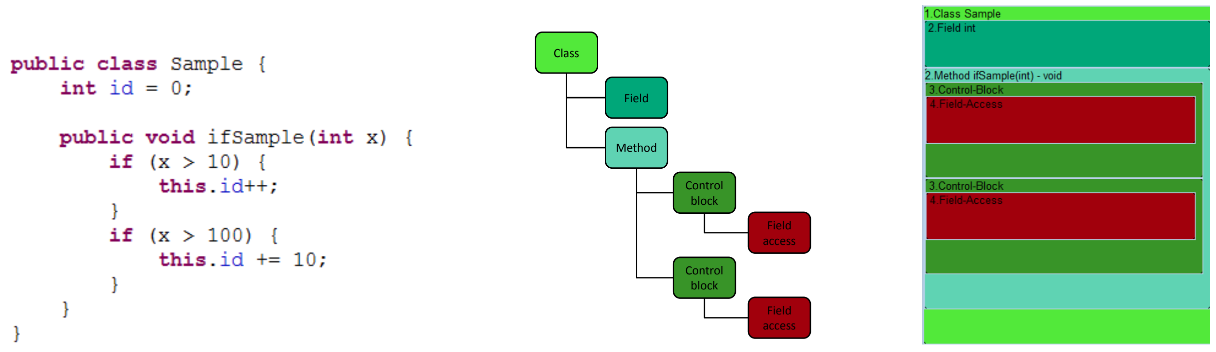
Code Feature	Example
Exception handling (rank = 1)	
Try Block	`try {...`
Catch Block	`catch (Exception e) {...`
Throw Statement	`throw new Exception();`
Local variable declarations (rank = 2)	
Local Variable	`int a = 5;`
Method invocations (rank = 3)	
Method Invocation	`a(0);`
Super-Method Invocation	`super.a(0);`
Constructor Invocation	`new A(0);`
Super-Constructor Invocation	`super(0);`
Field Access (rank = 4)	
Field Access	`this.a = 5;`
Super-Field Access	`super.a = 5;`
Iteration constructs (rank = 5)	
For-Loop	`for (i=1; i<n; i++) {...`
Enhanced For-Loop	`for (Int i : ints) {...`
While-Loop	`while (a > 5) {...`
Do-While-Loop	`do {...`
Flow control (rank = 6)	
Control Block	`if (a > 5) {...`
Switch Block	`switch (a) {...`
Conditional Expression	`...a > 5 ? 0 : 1...`
Case Block	`case 1: ...`
Return Statement	`return a;`
Break Statement	`break;`
Miscellaneous (unranked—always shown)	
Field	`int a;`
Anonymous Class Block	`return new A(){...`
Synchronized Block	`synchronized (a) {...`
New Class Instance	`A a = new A();`

2.2 Visualizing Code Features

We designed visualization elements as rectangular shapes so that they can be easily aligned and nested, thereby improving the user's ability to quickly glance at the visualization to perceive structure (a reminiscent visualization strategy has been used by the Path Projection system [11] to visualize call stacks). Each visualization element has a label that carries two pieces of information: the depth of the code feature in the parse tree and the type of code feature. Visualization elements also have a designated color. This enables users to quickly locate particular code features and it also enables expert users to quickly see particular recurring graphical patterns in the visualization (such as visual patterns generated by getter-and-setter methods).

The system extracts code features from the AST of the source code file of the active code editor in Eclipse. We parse the AST and perform a topology-preserving transformation that eliminates all nodes in the AST that are not code features (such as for example the `Body` nodes). This results in what we call a *code feature tree*. Figure 3 illustrates how a visualization is mapped from the original source code.

Visualization elements are nodes in the code feature tree. Visualization elements preserve the hierarchy of the code feature tree but all corresponding visualization elements are not necessarily shown to the user. An important design decision was to enable lightweight visualization. The visualization is meant to be used in tandem with the code editor and any other supporting views that are open in the user's IDE. Therefore, the visualization must be flexibly designed so that it can function effectively in a relatively small window (which may occasionally get resized by the user). It should also be designed to be relatively structurally stable in order to avoid distracting the programmer by complete layout reconfigurations. Another related design requirement was that users should never have to scroll within the visual-



```
public class Sample {
    int id = 0;

    public void ifSample(int x) {
        if (x > 10) {
            this.id++;
        }
        if (x > 100) {
            this.id += 10;
        }
    }
}
```

(a) Source code (b) Tree structure (c) Visualization

Figure 3. Source code, its tree representation, and its nested tree visualization.

ization window as this prevents users from quickly glancing at the visualization while they are working in the code editor.

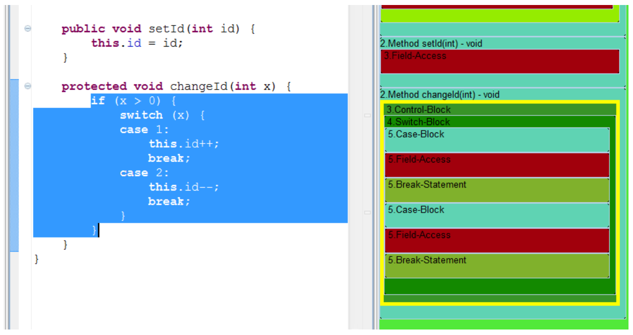
The visualization handles the problem with limited screen estate in a variety of ways. First, the visualization hides code features according to their ranking if there is not sufficient space to show all visualization elements.

Second, the visualization supports different zoom-levels. Zoom-levels are hints from the user that tells the system the granularity of the visualization the user is likely to be interested in. Figure 2 shows five different zoom levels for the source code file in Figure 1. The first zoom-level provides a general overview of the classes, class variables and methods of the file that is visualized. The remaining zoom levels visualize the code features of the file in increasingly higher resolution. Whether a code feature is shown for a particular zoom level depends on its rank and its nesting level in the code feature tree.

2.3 Editor Integration

LICV is tightly integrated with the active code editor in the Eclipse IDE. The system tracks the user's editing operations and updates the visualization in real-time with no perceptually noticeable delay as the user is typing or editing code. This is unlike other lightweight software visualization approaches (such as transient visualizations [8]), which need to be explicitly invoked by the user (however note that transient visualizations were designed for a different purpose than LICV).

It is often difficult to achieve a smooth novice-to-expert transisition for advanced development tools [18]. We conjecture that visualizations that automatically and continuously update themselves enable the visualization to be part of the user's "peripheral" vision. We suggest that such automatic code visualization can help a novice to gradually learn how the code visualization works. This can be an important feature as it is likely that some users do not wish to invest time into learning how code visualization works through a tutorial or similar training support.



```
    public void setId(int id) {
        this.id = id;
    }

    protected void changeId(int x) {
        if (x > 0) {
            switch (x) {
            case 1:
                this.id++;
                break;
            case 2:
                this.id--;
                break;
            }
        }
    }
}
```

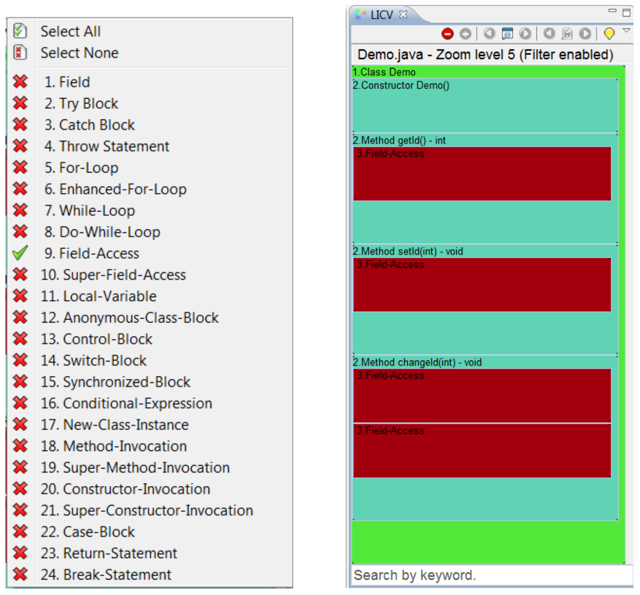
Figure 4. When the user selects code in the editor the system automatically indicates the corresponding code elements in the visualization with a yellow rectangle.

In order to support users retrieving details on demand, when the user clicks on a visualization element the system automatically locates the corresponding code in the active editor and scrolls the editor window so that the code becomes visible to the user. If the user selects code in the active editor the system will highlight the corresponding visualization elements. Figure 4 shows how selecting the `if` block in the active editor results in the system highlighting the corresponding `Control-Block` in the visualization window (including any nested elements that can fit within the current size of the window and are permissible according to the current zoom-level and filter settings of the visualization).

2.4 Filtering

Filtering enables the user to focus on the aspects of the code visualization that matters the most for a given task. LICV provides two different methods for filtering.

One way for users to filter is by using incremental search. At the bottom of the LICV window is a search field (see Figure 1). When the user types text into the search field the system performs a case-sensitive incremental search on the underlying code that generated the currently shown visualization. The system then only shows visualization elements


Select All
Select None

✗ 1. Field
✗ 2. Try Block
✗ 3. Catch Block
✗ 4. Throw Statement
✗ 5. For-Loop
✗ 6. Enhanced-For-Loop
✗ 7. While-Loop
✗ 8. Do-While-Loop
✓ 9. Field-Access
✗ 10. Super-Field-Access
✗ 11. Local-Variable
✗ 12. Anonymous-Class-Block
✗ 13. Control-Block
✗ 14. Switch-Block
✗ 15. Synchronized-Block
✗ 16. Conditional-Expression
✗ 17. New-Class-Instance
✗ 18. Method-Invocation
✗ 19. Super-Method-Invocation
✗ 20. Constructor-Invocation
✗ 21. Super-Constructor-Invocation
✗ 22. Case-Block
✗ 23. Return-Statement
✗ 24. Break-Statement

LICV
Demo.java - Zoom level 5 (Filter enabled)
1.Class Demo
2.Constructor Demo()
2.Method getId() - int
3.Field-Access
2.Method setId(int) - void
3.Field-Access
2.Method changeId(int) - void
3.Field-Access
3.Field-Access
Search by keyword.

(a) Filter (b) Result

Figure 5. Filtering the visualization so that it only includes `Field-Access` and code features that cannot be removed (classes, methods and constructors).

whose underlying code matches the incremental search query.

Filtering by incremental search enables the user to quickly focus on particular aspects of the code, such as for example all methods and instance variables that have `protected` access. In many cases the user only needs to type `pro` for the system to filter out any other visualization elements.

Another way for users to filter is to bring up a pull-down menu with all code features. The user can use this pull-down menu to visualize all, none, or particular code features. However, three code features are always shown regardless of filter settings: classes, methods and constructors. Figure 5 illustrates how the user has configured the filter so that `Field-Access` is the only optional code feature to be visualized.

3. Evaluation

Our central hypothesis is that LICV will aid programmers in performing a series of routine programming tasks. We predict the combination of lightweight visualization and tight integration with the IDE will enable experienced Eclipse users to perform a set of tasks faster than using the regular Eclipse IDE.

One relatively inexpensive yet reasonably rigorous approach to assess the initial viability of LICV is controlled experiments. This evaluation paradigm is not unusual when evaluating software engineering and programming tools (e.g. [6, 7]) when measurable objective functions have been identified, such as completion times, errors and users' preferences. However, controlled experiments also have certain limitations. In particular, in the context of software engineering, controlled experiments have been criticised for being "one-off" results that are seldom replicated [15].

To increase the validity of our findings we carried out the same experimental design twice with two different sets of participants (and slightly different setups). As we we will see, the second replication experiment confirmed the results of the first experiment.

3.1 Method

We evaluated LICV using a within-subjects experimental design. We compared LICV against the regular Eclipse IDE in four different experimental tasks. In other words, the independent variable was *Tool* with two levels: LICV and regular Eclipse. The order of these two conditions was balanced across the participants to avoid any learning effects.

We measured three dependent variables: completion time (in seconds), errors, and users' subjective preference on an ordinal scale (1 = minimum preference, 11 = maximum preference).

The four experimental tasks were chosen based on Java programmers' usage patterns when using the Eclipse IDE [16]. Similar tasks have also been used before, for example to evaluate fisheye visualization of Java source code in Eclipse [9]. We do not claim these four tasks are representative of everyday programming activities. As this was an initial formative evaluation we opted for constrained tasks that favor internal validity at the expense of external validity.

3.1.1 Task 1

Participants were provided with 280 lines of well-formatted Java code that described a single class. The code included comments. Participants were instructed to perform two subtasks by reading the code and attending to the visualization (in the condition that included the visualization).

The first subtask was to count the number of global (class) variables in the code. In the baseline condition this subtask could be achieved by either using Eclipse's *Outline View* or by directly reading the code. In the visualization condition this subtask could also be achieved by counting the number of corresponding elements in the visualization.

The second subtask was to locate all the class constructors in the code and then to give the line number of the first line for each constructor. In the baseline condition this subtask could be achieved by either using Eclipse's *Outline View* or by directly reading the code. In the visualization condition this subtask could also be achieved by clicking on all elements in the visualization that contained the keyword "Constructor". This action triggered the system to automatically display the corresponding code in the Eclipse code editor window (scrolling the code editor window, if necessary).

3.1.2 Task 2

Participants were provided with well-formatted Java code and requested to focus on a block of nested loops consisting of 104 lines. Participants were instructed to answer whether six statements about the code were true or false.

The statements were the following:

1. The outermost loop is a while loop but not a do-while loop.
2. The outermost loop contains a break statement.
3. The second nested level has a do-while loop.
4. The second nested level has more than one loop.

For the following questions, "nested level" of the outermost loop is 1.

5. The deepest nested level of this block is 4.
6. The deepest loop is an enhanced for-loop.

In the baseline condition this task could be achieved by using the search function or by directly reading the code. In the visualization condition this could also be achieved by highlighting the code and then applying filters in the code visualization so that only loops and break statements are visualized.

3.1.3 Task 3

Participants were provided a file with 890 lines of well-formatted Java code. Participants were instructed to identify and resolve a runtime exception caused by a given method call statement that appeared multiple times in the file. Participants were instructed to search and check each suspected statement in order to find the error that causes the runtime exception. Once the error was detected they were asked to fix it. This required adding a single line of code. Two separate files consisting of 890 lines each were used for this task as the same error cannot be present in both conditions. The order of these files were counterbalanced to avoid any potentially confounding crossover effects.

In the baseline condition participants could achieve this by searching, filtering and locating the provided statement using the search function of Eclipse and thereafter inspect (and possibly modify) the code in the editor directly. In the visualization condition participants could also achieve this by first setting the code feature filter to only visualize method invocations and then using incremental search to locate all the suspected method call statements. Once located the participant would inspect (and possibly modify) the code in the editor directly.

3.1.4 Task 4

Participants were provided a large file consisting of well-formatted Java code and instructed to insert a single statement just before the return statement in all methods in the file. Participants were told to type the statement the first time and then copy and paste the statement for all remaining instances.

In the baseline condition participants could achieve this by using Eclipse's search function to locate the correct lines in the code and thereafter add the required statement. In the visualization condition participants could also achieve this by first using the code feature filter to filter out all code features in the visualization except for return statements. Thereafter they could click on elements in the visualization that contained the keyword "Return-Statement". This action makes the system locate the corresponding return statement in the code editor and highlight it. Following this action they could then add the required statement right before the return statement.

3.2 Experiment 1: Lab Study

In the first experiment we recruited nine computer science students via convenience sampling. The participants had on average four years of experience with Java ($sd = 1.6$) and three years of experience with the Eclipse IDE for Java ($sd = 1.6$). The least experienced participant had one year of experience and the most experienced participant had six years of experience with Java and Eclipse.

Participants used the Eclipse IDE version 3.6.1 in the experiment. To provide a consistence development platform for all the participants, all the experiments were conducted on a MacBook Pro with Eclipse IDE for Java Developers 3.6.1 running on Mac OS X 10.6.7. The experiment was conducted in a quiet office. The Eclipse keyboard bindings were adjusted to match the participants' prior experience (a Windows user would use Windows key-bindings, and so on).

Before testing, participants were demonstrated how LICV works within Eclipse and offered a few minutes to familiarize themselves with the laptop and the Eclipse environment.

Thereafter they were asked to complete the four tasks we described in the previous section.

3.2.1 Results

We analyzed errors and completion times using repeated measures analysis of variance at the $\alpha = 0.05$ significance level.

The mean errors were (LICV vs. baseline): 0 vs. 0.6 for task 1, 0.2 vs. 0.9 for task 2, 0 vs. 0.7 for task 3 and 0 vs. 0 for task 4. Only the mean error difference in task 2 was statistically significant ($F_{1,8}$ = 16.000, η_p^2 = 0.667, $p <$ 0.005).

Completion time was measured as the number of seconds participants spent on the task. The time interval was from when the participants opened the Java code file for the task to when the participants stated that they had completed the task. As completion times (similar to response times) are heavily skewed we log-transformed them before the statistical analysis.

As can be seen in Figure 6, participants were substantially faster in tackling tasks 1, 2 and 4 using LICV. Mean comple-

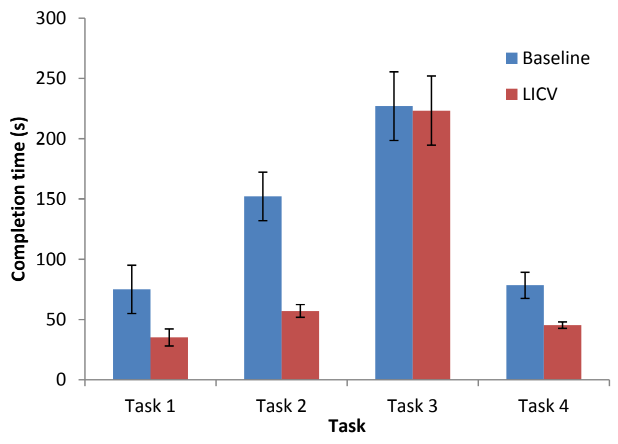


Figure 6. Completion times (in seconds) and 95% confidence intervals as a function of the task in experiment 1.

tion times were (LICV vs. baseline): 35 vs. 75, 57 vs. 152, 223 vs. 227 and 45 vs. 78 seconds. The mean completion time differences in task 1, task 2 and task 4 were statistically significant ($F_{1,8}$ = 6.761, 37.602 and 16.552, η_p^2 = 0.458, 0.825 and 0.674, $p <$ 0.05, 0.0001 and 0.005).

Participants were asked to rate their preferences for both the baseline and the visualization method when they solved the tasks. Participants rated their preference on an 11-point Likert scale (1 = minimum preference, 11 = maximum preference). Median preferences were (LICV vs. baseline): 8 vs. 5 for task 1, 9 vs. 3 for task 2, 8 vs. 5 for task 3 and 7 vs. 7 for task 4. Friedman's test at the $\alpha = 0.05$ significance level showed that the differences in preferences were statistically significant for tasks 1–3 (χ^2 = 4.500, 5.444 and 5.444, *df* = 1).

3.3 Evaluation 2: Replication with Remote Users

In the replication experiment we recruited seven participants via convenience sampling. The participants had on average five years of experience with Java ($sd = 1.6$) and three years of experience with the Eclipse IDE for Java ($sd = 1.3$). The least experienced participant had three years of experience with Java and one year of experience with Eclipse, while the most experienced participant had eight years of experience with Java and five years of experience with Eclipse.

Apart from participants performing the tasks on their own laptops, the procedure was identical to the first experiment. Participants were instructed and supervised via a videoconferencing facility.

3.3.1 Results

Participants made very few errors. The mean errors were (LICV vs. baseline): 0 vs. 0.14 for task 1, 0 vs. 0.57 for task 2, 0.71 vs. 1.0 for task 3 and 0 vs. 0 for task 4. None of the differences were statistically significant.

We found that LICV dramatically reduced completion times for tasks 1, 2 and 4. Mean completion times were (LICV vs. baseline): 23 vs. 98, 49 vs. 140, 208 vs. 270 and 38 vs. 75 seconds. The mean completion time differences in task 1, task 2 and task 4 were statistically significant ($F_{1,6}$ = 22.373, 62.636 and 25.670, η_p^2 = 0.789, 0.913 and 0.811, p < 0.005, 0.0001 and 0.005).

We found that users significantly preferred to use LICV. Median preferences were (LICV vs. baseline): 9 vs. 5 for task 1, 9 vs. 2 for task 2, 8 vs. 4 for task 3 and 8 vs. 5 for task 4. Friedman's test at the $\alpha = 0.05$ significance level showed that the differences in preferences were statistically significant for all four tasks (χ^2 = 7.000, *df* = 1).

In summary, the replication experiment verified the results we obtained in the first experiment. LICV significantly reduces completion times for tasks 1, 2 and 4. Furthermore, users significantly prefer to use LICV.

4. Discussion

The two experiments demonstrated that LICV resulted in significantly faster completion times for three out of the four experimental tasks we investigated. Moreover, users preferred LICV over the regular Eclipse IDE, even though they had at least one year of experience with it and had only been exposed to LICV for a very short amount of time.

However, we caution against overinterpreting the experimental results. A limitation with the evaluation is the particular selection of experimental tasks. LICV does not appear to have helped much in reducing completion times for task 3, a debugging task, which is the task that took the participants the longest amount of time. Therefore, it is possible that LICV is not as efficient for debugging as in assisting with basic program comprehension and routine program editing. This is not surprising as debugging is a complex programming activity and as a consequence several advanced systems have been developed to support debugging tasks, such as Whyline [12].

Another limitation is the fact that we assessed LICV's performance in controlled experiments under a limited amount of time. While such studies are useful in verifying initial hypotheses, they tell us little about actual use in real tasks. Such insights can only be captured via longitudinal in-situ studies, for example longitudinal evaluations based on experience sampling and log studies [9]. The two experiments are neither large enough, nor diverse enough in terms of the tasks they explore, to conclusively state whether LICV is successful in aiding programmers with routine tasks. Finally, while we have evaluated a few points in the possible LICV design space there are certainly many design parameters that can be explored in future work, such as the choice of code features, means of prioritizing code features, and visualization strategies for code features.

5. Conclusions

In this paper we have presented Lightweight Integrated Code Visualization (LICV) as an approach for improving programmers' productivity. LICV is based on four design principles: 1) it is a secondary interface; 2) it is lightweight; 3) it follows the visual information-seeking mantra; and 4) it supports a smooth novice-to-expert transition.

We implemented a LICV tool as a plug-in for the Eclipse IDE for Java Developers. We evaluated our LICV tool in two controlled experiments and found that LICV significantly reduced participants' completion times by nearly 50% for three out of four tasks. Furthermore, participants also significantly preferred using LICV to perform the tasks. LICV and regular Eclipse performed similar in terms of errors.

We hope the positive *indicative* results in this paper will lead to further design explorations of lightweight integrated code visualization to support existing programming environments and work practices.

References

[1] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, pages 455–464. ACM Press, 2010.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2009)*, pages 213–222. ACM Press, 2009.

[3] R. DeLine and K. Rowan. Code Canvas: Zooming towards better development environments. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, pages 207–210. ACM Press, 2010.

[4] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger Canvas: Industrial experience with the Code Bubbles paradigm. In *Proceedings of the International Conference on Software Engineering (ICSE 2012)*, pages 1064–1073. IEEE Press, 2012.

[5] S. Ducasse and M. Lanza. The Class Blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[6] A. Dunsmore, M. Roper, and M. Wood. The development and evaluation of three diverse techniques for object-oriented code inspection. *IEEE Transactions on Software Engineering*, 29 (8):677–686, 2003.

[7] M. R. Jakobsen and K. Hornbæk. Evaluating a fisheye view of source code. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2006)*, pages 377–386. ACM Press, 2006.

[8] M. R. Jakobsen and K. Hornbæk. Transient visualizations. In *Proceedings of the Australasian Conference on Computer-Human Interaction (OzCHI 2007)*, pages 69–76. ACM Press, 2007.

[9] M. R. Jakobsen and K. Hornbæk. Fisheyes in the field: using method triangulation to study the adoption and use of a source code visualization. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2009)*, pages 1579–1588. ACM Press, 2009.

[10] J. G. Kemeny. Mathematics without numbers. *Daedalus*, 88 (4):577–591, 1959.

[11] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path projection for user-centered static analysis tools. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2008)*, pages 57–63. ACM Press, 2008.

[12] A. J. Ko and B. A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2004)*, pages 151–158. ACM Press, 2004.

[13] M. Lanza and S. Ducasse. Polymetric Views—A lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 48–61. ACM Press, 2005.

[15] J. Miller. Replicating software engineering experiments: A poisoned chalice or the Holy Grail. *Information and Software Technology*, 47:233–244, 2005.

[16] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[17] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Grapacc: A graph-based pattern-oriented, context-sensitive code completion tool. In *Proceedings of the International Conference on Software Engineering (ICSE 2012)*, pages 1407–1410. IEEE Press, 2012.

[18] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38 (6):33–44, 1995.

[19] R. Robbes, S. Ducasse, and M. Lanza. Microprints: A pixel-based semantically rich visualization of methods. In *Proceedings of International Smalltalk Conference*, pages 131–157, 2005.

[20] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343. IEEE Press, 1996.

[21] J. Stylos, B. A. Myers, and Z. Yang. Jadeite: Improving API documentation using usage information. In *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI 2009)*, pages 4429–4434. ACM Press, 2009.

[22] M. Weiser and J. S. Brown. Designing calm technology. *PowerGrid Journal*, Version 1.01, 1996. `http://www.powergrid.com/1.01/calmtech-essence.html`.

[23] H. P. Young and A. Levenglick. A consistent extension of Condorcet's election principle. *SIAM Journal on Applied Mathematics*, 35(2):285–300, 1978.