Method-flow: A software development visualisation technique for multi-dimensional program navigation and composition

Daniel Bradley
School of Information Technology and Electrical Engineering,
The University of Queensland, Brisbane, Australia, 4072
daniel.bradley@acm.org

Abstract

Mainstream *Integrated Development Environments* (IDEs) predominantly focus on using the *file hierarchy* for navigating software. During navigation each file is obscured by the next leading to disorientation and a loss of task awareness. This paper presents the *method-flow* software visualisation technique, which supports software navigation and composition by displaying traversed methods in adjacent code editor columns. The technique has now been implemented within a programming environment called Visuocode.

Keywords

software development visualisation, software navigation, software composition, software exploration, programming dimensions, program slicing, visuospatial programming

1 Introduction

Mainstream *Integrated Development Environments* (IDEs) predominantly focus on using the *file hierarchy* as a means of navigating software. While some environments support hyperlinked method navigation, source code remains presented in text editors that show the contents of the entire source code file. The programmer is limited to only viewing a single method unless, by luck or design, an associated method is placed either before, or after, that method in the file.

The call-graph of programs written in object-oriented languages such as Java are usually orthogonal to the file structure of their source code because each class is stored in a separate text file. In order to understand what is happening within a program, programmers often navigate the call-graph of a program, which necessitates navigating through a potentially large number of different source code files. This often results in programmers becoming disoriented – they forget what methods calls brought them to where they are – and losing task awareness – they forget why they started performing their current task (de Alwis & Murphy, 2006) [1]. This occurs because when another method is navigated to, the editor is either scrolled, replaced, or obscured. This often results in the programmer switching back and forth between different files – a behaviour referred to as *thrashing* (de Alwis & Murphy, 2006) [1].

Programming environments also inflict *class creation overhead* – the cost of the interruption when a programmer creates a new class. This cost reflects not only time lost, but also the distraction caused to the programmer. It is hypothesised that this overhead, combined with the difficulty of seeing both contexts at once, encourages programmers to avoid separating out code into methods or new classes when a new level of abstraction is appropriate. The result is that classes and methods may be larger than appropriate compromising software structure.

Text editor based programming environments have previously been described as one-dimensional (Myers, 1986) [2]. To mitigate the problems described above, programming environments need to support additional *programming dimensions*. Such an environment would allow the programmer to predominantly navigate between methods and classes without resorting to auxiliary navigation.

This paper introduces the *method-flow*[1] software visualisation technique. Method-flow allows the programer to navigate the call-graph of a program using adjacent editor columns. During development, adjacent columns can be used to create new software structures without disrupting the existing programming context, and minimising interruption.

The structure of this paper is as follows: Section 2 further describes the multi-dimensional aspects of software; Section 3 introduces the method-flow software visualisation technique; Section 4 describes the Visuocode programming environment; Section 5 shares findings of initial informal evaluation; Section 6 compares and contrasts method-flow to related work; and Section 7 provides a summary.

2 Programming dimensions

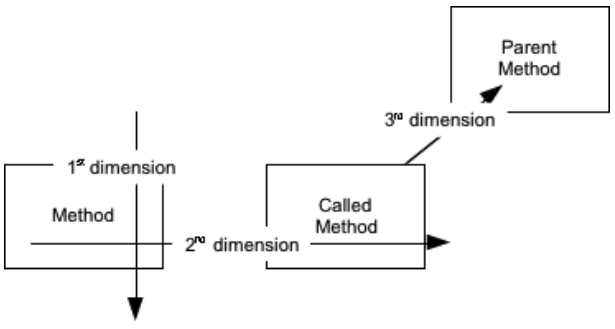

Figure 1: Three programming dimensions

2.1 The 1st dimension

If a text editor can be described as being one-dimensional due to only being able to see the state-

[1]This technique has been formerly referred to as 'code-flow'

Cite as bibtex @misc:
Bradley, Daniel R. (2012) Method-flow: A software development visualisation technique for multi-dimensional program navigation and composition. http://www.danielbradley.org/publications/pdf/2012Bradley-1.pdf. October, 2012.

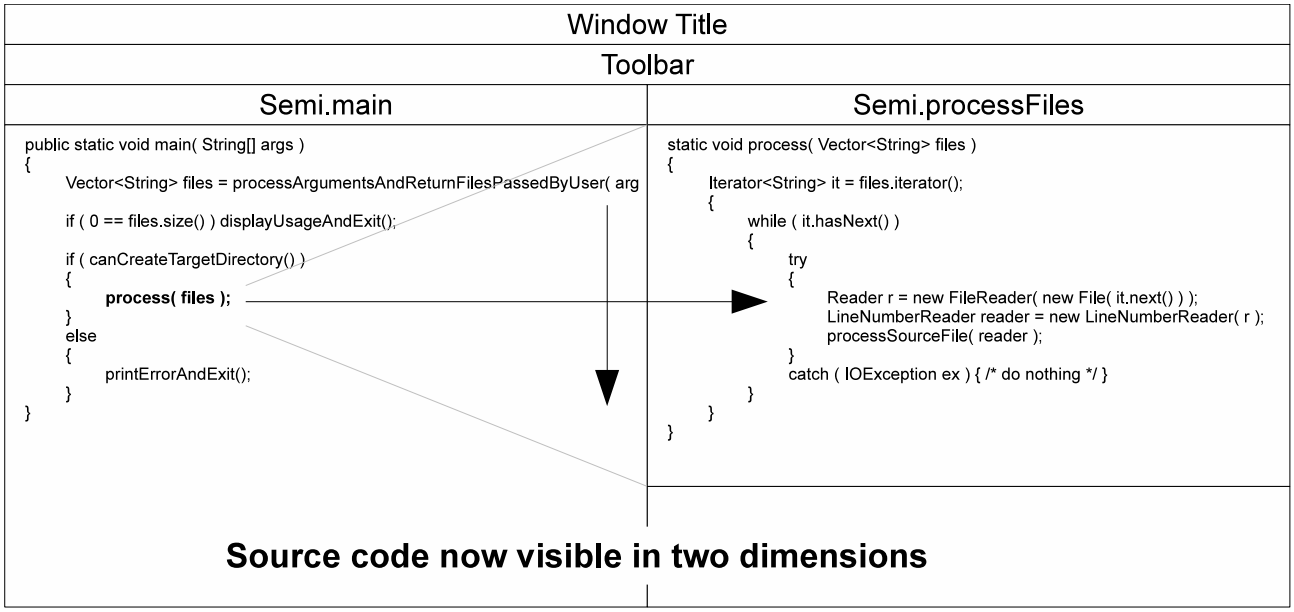

Figure 2: A development environment that supports two programming dimensions

ments of one chosen method at a time, it follows that a program without procedures can be considered to be one-dimensional because all statements are executed within the same *layer of abstraction*.

2.2 The 2nd dimension

Procedures and functions add another programming dimension as they enable *layers of abstraction* – subprocedures allow an arbitrary number of statements to be considered as one statement. For example, the following statement abstracts the mathematical calculations required to return the square-root of '30':

```
x = squareRoot( 30 );
```

Similarly, in an object-oriented programming languages methods provide an extra layer of abstraction and therefore are also considered to represent the second dimension. For example, the following statement provides a similar abstraction to the previous example:

```
x = anInteger.squareRoot();
```

2.3 The 3rd dimension

A third dimension, however, is added through class inheritance as overriding methods may possibly also call the method they overrode within an ancestor class. When a statement instantiates an object by calling a constructor, at one level of abstraction an object is returned that is appropriately initialised. At a lower level of abstraction, the constructor first calls its parent's constructor to initialise the state of any data members defined within the parent, then executes its own statements to initialise its own state and potentially override the state of the parent class. During software composition, the programmer is often responsible for both of these layers of abstraction. It is important that the programmer fully understands what is happening in the calling context – where the object is being instantiated – and in the called context – the initialisation of the object.

3 The method-flow visualisation technique

It is often suggested that the programmer may simply manually arrange editors to see multiple programming dimensions at once. This proves more difficult than one might expect due to modern IDEs such as *Eclipse* and *X-Code* constraining how files are presented within the workspace window.

X-Code 4 provides *companion editors*, which can automatically change based on the file in the primary editor – for example, it may be set to display the header file of a C-style language source file. In practice, however, the contents of this editor is often manually selected.

In Eclipse each file is opened in a tabbed editor. While a tab may be dragged to the side of the editor area to split the screen between two files screen space soon becomes a limiting factor as the editor area is constrained. Also another editor can only be used if the method is located in a different file.

Even with stand-alone text editors this strategy proves unworkable as often a programmer will want to browse through multiple layers of abstraction.

3.1 Supporting the 2nd dimension

A simple technique for supporting the second programming dimension would be to allow the programmer to cause another editor to be opened beside the existing editor. As shown in Figure 2, following a hyperlinked method call would cause the called method to be displayed adjacent to the existing method. Crucially, however, an arbitrary number of editors should be able to be shown by scrolling the leftmost editors off the screen. The benefit of this technique is that a programmer can refer to an arbitrary number of methods without disrupting the existing programming context.

When a new method is required, the programming environment would be able to recognise a non-resolving method-call and automatically create and display the new method skeleton.

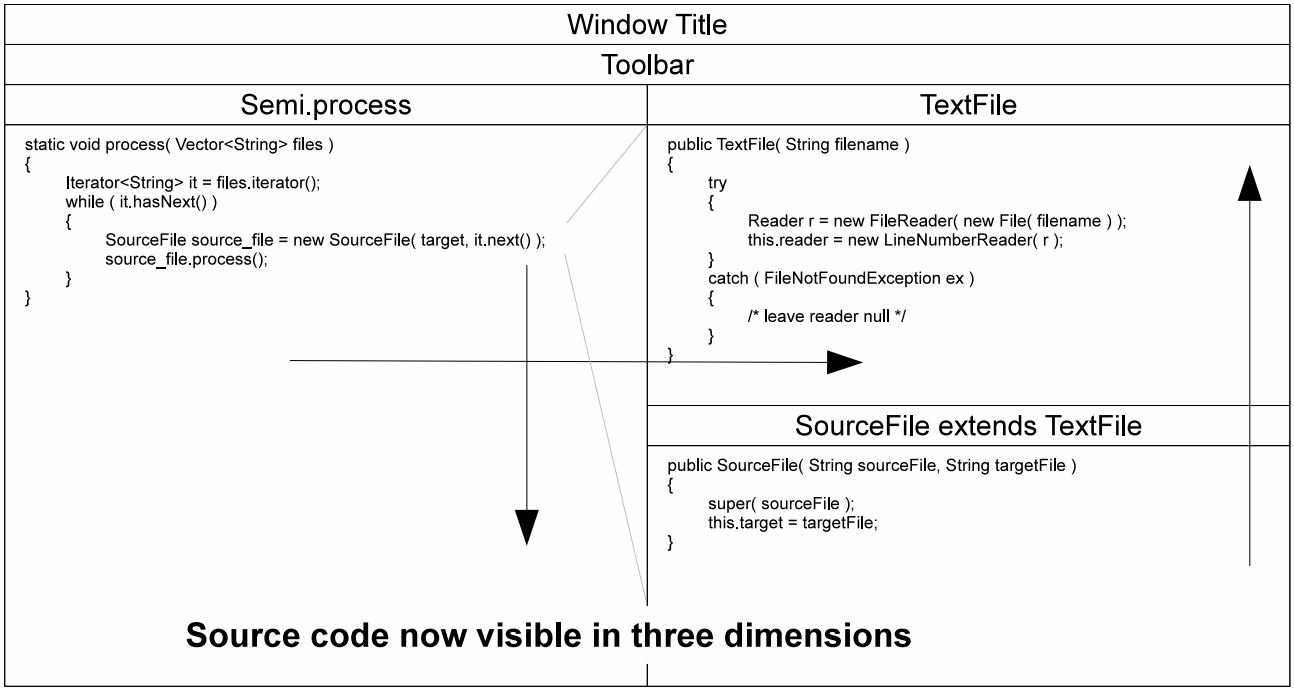

Figure 3: A development environment that supports three programming dimensions

3.2 Supporting the 3rd dimension

The third programming dimension is brought about through method calls to overridden methods and constructors in an ancestor class. A method of visualising such method calls is to identify the target method and display it above the called method. Figure 3 shows an example with two constructors – *SourceFile* and *TextFile* – however an arbitrary number of such methods could also be stacked upon each other. The benefit of this representation is that (in the case of constructors at least) the execution of statements should proceed from the top of the screen down to the bottom of the screen.

Alternatively, such method calls could be displayed in another editor column. However, doing so would consume more screen real estate than otherwise, and that column editor would also be replaced if another method hyperlink were followed.

3.3 Method-flow

'Method-flow' is the name we have given to this technique of software navigation. Formally: The *method-flow* software visualisation technique explicitly supports software navigation and composition by allowing the programmer to traverse a method call-graph using adjacent editor columns.

In each editor column, method calls are presented as hyperlinks. If a hyperlink is followed, a new column is stacked to the right of that column. As more text columns are added, the leftmost columns scroll off the left side of the screen.

3.4 Program composition using method-flow

During program composition, when a programmer decides to create a new method they have a similar problem: if creating a method in a different class they must open that file in a new editor, which either replaces or obscures the current editor; or if creating a method in the current class they must scroll their editor to the desired position for the method, then return when completed.

When using method-flow, a new class or method can be displayed adjacent to the existing programming context in a new editor column.

4 Visuocode

The current implementation, called Visuocode, now supports program composition and is available for download [2]. The programming environment consists of two types of window the *workspace manager* and associated *flow windows*. The following terminology was intentionally chosen to be analogous to terminology used by other programming environments.

4.1 The workspace manager

The workspace manager (see Figure 4) allows the programmer to manage a collection of projects. Each project represents a collection of one or more Java packages that might be bundled together within a single Java *jar* archive for distribution. A simple workspace might link to only one project while others may link to many. Projects may be shared by any number of workspaces.

When a project is added to the workspace manager its Java source files are parsed and a tree representing its class hierarchy is added beneath the "Workspace Projects" node. Following a class or method name will cause a flow window to appear.

4.2 The flow window

Initially, the flow window contains a single column editor. Each column editor contains a *class attributes* area and a *method editor* area. The class attributes area contains an editable class field, as well as three editable field lists that allow imports, enumerations, and class members to be altered.

[2] www.visuocode.com

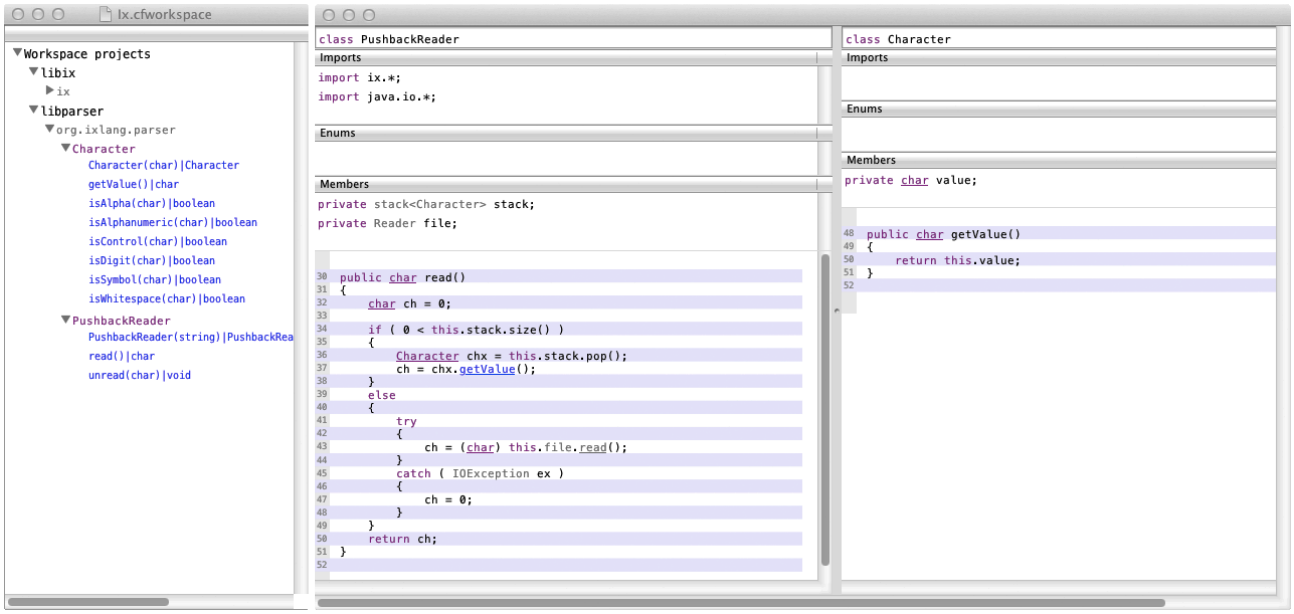

Figure 4: Visuocode consists of two types of window the *workspace manager*, and the *flow window*

The method editor area contains a stack of one or more method editors. If the column editor was opened by clicking a class name in the workspace manager, there will be a method editor for each of the class's methods ordered alphabetically.

Within method editors, resolving method calls are represented as blue hyperlinks. When followed, an editor column for that method is inserted to the right of the existing column shifting all existing columns left. As all columns sit within a scroll-pane, the programmer can scroll back to the left to view obscured column editors.

4.3 Support for software creation

Method calls that fail to resolve are represented as red hyperlinks. When followed, an editor column containing a method skeleton is displayed. The method call is analysed to determine in which class the new method should be inserted. For example, a non-resolving implicit method call, or a method call explicitly invoked on *this*, would be inserted into the current class, e.g.,

```
int x = getValue();
```

or

```
int x = this.getValue();
```

Where a non-existent method is invoked upon a specific object, it would be inserted into the class corresponding to that object. For example, assuming that the *getArea* method has not been defined in the following code snippet, following the method call would display a column editor for the Shape class that contains an appropriate method skeleton, e.g.,

```
Shape aShape = new Shape();

int x = aShape.getArea();
```

4.4 Architecture

Visuocode has been implemented to take advantage of the Mac OS Cocoa programming API. The user interface has been implemented in Objective-C++ and depends on a variety of libraries written in Objective-C++, C++, and C.

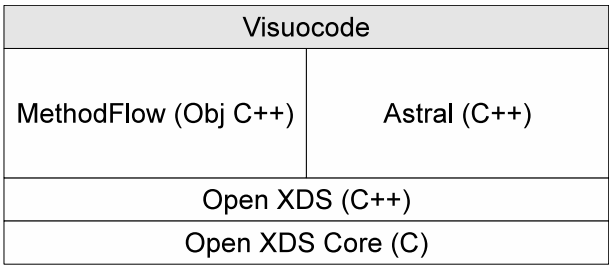

Figure 5: Visuocode architecture

Figure 5 shows the architecture of the Visuocode application. *Open XDS* provides a set of cross-platform class libraries for developing C++ programs similar to the Java class library. *Astral* includes libraries that are used for source tokenization and parsing, as well as source code management. *MethodFlow* consists of a *model* library that wraps the Astral libraries and various *view* libraries. Visuocode, itself, is little more than a wrapper generated by X-Code.

4.5 Limitations

The following limitations have been identified in the current implementation:

1. Visuocode only supports navigation in the 2nd dimension – it does not support showing ancestor methods within the same editor column.
2. Currently, classes need to be manually created using a dialog box. In the near future classes will be able to be instantiated by click non-resolving class types.
3. The system currently supports 'simple' enumerations, however, class-like enumerations are not currently supported.

4. Due to the system using a form of static analysis, the system is currently unable to identify the runtime type of polymorphic objects.

5 Evaluation

The Visuocode programming environment has been informally evaluated in preparation for an upcoming study. While initial use of the system has informally validated the concept, it has also highlighted the following problems that will need to be addressed.

5.1 Strangeness

Because the Visuocode environment represents code at a higher level of abstraction to file-based editors there is a feeling of being "out-of-control". This feeling lessens after a short period of time as the programmer is able to concentrate on programming instead of the environment. This feeling will need to be taken into account when planning participant training for future studies.

5.2 Lack of incremental compilation

Programmers have become accustomed to integrated compilation, which allows the environment to highlight which statements contain syntactic errors. While Visuocode parses the source code in order to populate its own internal data representation, it does not explicitly provide feedback to the user regarding incorrect syntax. It is expected that code will be compiled using an external application or command line tool. To aid problem detection, line-numbers have been added to method editors that allow the output from external tools to be cross-referenced with the Visuocode display.

5.3 Inability to identify runtime type of polymorphic objects

When an object is passed to a method, it is assumed to be of the type indicated in the method signature when it may actually be a sub-type. Due to Visuocode using a form of static analysis it has no knowledge of how the passed object was initially instantiated. Therefore, when a method call on such an object is followed it may cause an incorrect column editor to be displayed. As this can confuse and distract the programmer in the future the system will be modified to track the type of instantiated objects through method calls.

5.4 Inability to navigate juxtaposed call-graphs

A requested feature is for the programmer to be able to also navigate "backwards" up juxtaposed call-graphs by being shown all methods that call the current method. It is currently being considered how best to represent this in the Visuocode interface.

6 Related work

The method-flow visualisation technique has previously been implemented as a software exploration tool called *CodeFlow SE*. This tool, which was developed in the Java programming language, is available for download from the *code-flow* website.[3]

[3] www.code-flow.com

6.1 Program slicing

Conceptually the method-flow visualisation technique may be likened to a visual form of *program slicing* (Weiser, 1981) [3]. While originally program slicing referred to the identification of the subset of a program that generated a particular output, in the software visualisation literature *slicing* often refers to a logical subset of a program that is related to a specific statement. Various tools have been developed that represent this information. The *Ghinsu* tool uses a system dependence graph to highlight code statements that are related to a specified statement (Livadas & Alden, 1993) [4]. The *SeeSlice* tool presents source code as "code-thumbnails" and emphasises the code that belongs to a particular slice (Eick, Steffen & Sumner Jr, 1992) [5].

Unlike these tools, method-flow requires the user to manually select which methods along a branch of a call graph are shown.

6.2 Software exploration

A key aspect of method-flow is the ability to navigate down a call-graph. Software exploration tools provide similar functionality, however such tools usually provide an alternative representation such as a stylised UML-style diagram (Rigi, SHriMP, Rolo) (Mueller & Klashinsky, 1988) [6], (Storey et al., 2002) [7], (Sinha, Karger & Miller) [8], or a graph view (Whorf, Visual Call Graph) (Brade et al., 1992) [9], (Bohnet & Döllner, 2006) [10]. The user is then able to 'drill down' to the statement level by opening a source code viewer.

Such tools are often intended to aid the understanding of large software systems during maintenance, and may also prune the presented graph using slicing techniques. In contrast, method-flow is intended for use during software development and is specifically intended to allow the comparison of source code between methods along a branch of a program's call-graph.

6.3 Programming environments

6.3.1 In-lined code

Fluid Source Code Views allows the contents of called methods to be shown inline within an existing context (Desmond, Storey & Exton, 2006) [11]. This work and method-flow share a similar aim – to view the statements of a lower level of abstraction within the current context. However, a key difference between the two representations is that method-flow preserves the original programming context while the other does not as a single method call is replaced with multiple statements.

6.3.2 Block-emphasising environments

Emphasising the display of syntactic blocks (e.g., methods) rather than entire files is not a new concept. The *Pecan* system (Reiss, 1984) [12] provides a number of different views including a generated flow chart of functions, a syntax directed editor, and a data-structure view.

The *UQ Star* system similarly provides displays that focused on procedure blocks (Welsh, Broom & Kiong, 1991) [13].

Method-flow is differentiated from these systems as it allows an arbitrary number of column editors to be stacked adjacent to each other within a scroll-pane.

6.3.3 Desktop metaphors

Code Thumbnails represents source code as a "thumbnail image of the entire document" in either a sidebar associated with a specific file or within a *Code Thumbnail Desktop* that presents such a view for all files (DeLine et al., 2006) [14]. This concept was later extended as *Code Canvas* which extends the Code Thumbnail Desktop concept to provide a zoomable interface that allows the programmer to zoom out to see a fixed spatial representation of a software code base (a code map) or zoom in using a *semantic zoom* to show different amounts of information at different zoom points.

In contrast to method-flow, the Code Canvas system supports viewing multiple levels of abstraction by zooming in and out of software systems, whereas method-flow allows the direct navigation between linked methods.

Code bubbles is an impressive environment that displays methods in *bubbles* upon a "continuous virtual screen", which may be scrolled to either the left or right (Bragdon et al., 2010) [15]. Like method-flow, methods are represented in individual editors. Called methods may be manually "budded-off" and arranged in the proximity of the originating bubble allowing a programmer to view multiple levels of abstraction at once – in this way a chain of method bubbles may be constructed. Due to the obvious space constraints related to displaying multiple bubbles, Code Bubbles implements code reflow and elision, which breaks statements at intelligent points to minimise disruption to comprehension.

Apart from the use of the desktop metaphor, a key contrast between method-flow and Code Bubbles is that (like Code Canvas) Code Bubbles requires that budded-off bubbles be manually arranged on the desktop. Bubbles may also be arranged in scattered groups related to different tasks with the aim of allowing the preservation of spatial memory. In contrast, the method-flow visualisation technique has been specifically designed to not need manual arrangement of its column editors. Also a key intention of method-flow is that the appearance of method source code is not altered by techniques such as reflowing, or code elision.

7 Summary

This paper has introduced the method-flow software visualisation technique, which provides a novel method for a programmer to view multiple programming dimensions during software development. The technique supports software navigation that maintains the programmer's existing context and also allows for the convenient composition of new software structures. An implementation, Visuocode, which is available to download has been described, as well as the findings of an initial informal evaluation. Method-flow has also been compared and contrasted against related work.

Acknowledgements

I would like to thank my supervisor Dr Ian Hayes who has provided valuable feedback and suggestions on the drafts of this paper.

References

[1] B. de Alwis and G. Murphy, "Using visual momentum to explain disorientation in the Eclipse IDE," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 51 –54, September 2006.

[2] B. Myers, "Visual programming, programming by example, and program visualization: a taxonomy," in *Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pp. 59–66, ACM New York, NY, USA, 1986.

[3] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, (Piscataway, NJ, USA), pp. 439–449, IEEE Press, 1981.

[4] P. Livadas and S. Alden, "A toolset for program understanding," in *Proceedings of the 1993 IEEE Second Workshop on Program Comprehension*, pp. 110–118, 1993.

[5] S. Eick, J. Steffen, and E. Sumner Jr, "Seesoft-a tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[6] H. A. Mueller and K. Klashinsky, "Rigi — a system for programming-in-the-large," in *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, (Los Alamitos, CA, USA), pp. 80–86, IEEE Computer Society Press, 1988.

[7] M. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, "SHriMP views: an interactive environment for information visualization and navigation," in *Conference on Human Factors in Computing Systems*, pp. 520–521, ACM New York, NY, USA, 2002.

[8] V. Sinha, D. Karger, and R. Miller, "Relo: Helping users manage context during interactive exploratory visualization of large codebases," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 187 –194, sept. 2006.

[9] K. Brade, M. Guzdial, M. Steckel, and E. Soloway, "Whorf: a visualization tool for software maintenance," in *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pp. 148–154, 1992.

[10] J. Bohnet and J. D"ollner, "Visual exploration of function call graphs for feature location in complex software systems," in *Proceedings of the 2006 ACM symposium on Software visualization*, pp. 95–104, ACM New York, NY, USA, 2006.

[11] M. Desmond, M. Storey, and C. Exton, "Fluid Source Code Views," in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 260–263, IEEE Computer Society Washington, DC, USA, 2006.

[12] S. Reiss, "Graphical program development with PECAN program development systems," *ACM SIGPLAN Notices*, vol. 19, no. 5, pp. 30–41, 1984.

[13] J. Welsh, B. Broom, and D. Kiong, "A design rationale for a language-based editor," *Software: Practice and Experience*, vol. 21, no. 9, pp. 923–948, 1991.

[14] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson, "Code thumbnails: Using spatial memory to navigate source code," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 11–18, IEEE, 2006.

[15] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. 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*, CHI '10, (New York, NY, USA), pp. 2503–2512, ACM, 2010.