Development Tools

Devashish Jasani
Software Systems Engineering, RWTH Aachen, Germany
devashish.jasani@rwth-aachen.de

Bharath Rangaraj
Software Systems Engineering, RWTH Aachen, Germany
bharath.rangaraj@rwth-aachen.de

ABSTRACT

Developers use various IDEs as per their comfort in order to produce and maintain code. Software maintenance is a very time consuming task as it requires a developer to efficiently navigate through the various dependencies and comprehend it. Doing this without any extra aid is a tedious job. Code comprehension and code navigation are two of many important processes that a developer undertakes during software maintenance tasks. Integrated Development Environments (IDEs) can aid these processes. This can be done by providing different visualisations of code, widgets and windows and abstracting non-essential information from the Interface. In essence, IDEs should cater to task context by modelling developer strategies. User studies can analyse developer's navigation behaviour and show how this task context can be analysed. Some IDEs, tools, plugins implement these concepts to help developers comprehend and navigate through source code effectively and efficiently. Empirical studies indicate an increased developer productivity, when using such tools.

Author Keywords
Code maintenance, code comprehension, code navigation, developer strategies, task context.

INTRODUCTION

Developer spend a huge fraction of time, maintaing existing code. This includes adding new feature, fixing bugs and code refactoring [1]. These activities accounts for almost 70% of the total expenses for a software project [2].

Figure 1. shows the fraction of time devoted by developers for different tasks during a typical software project. As time progresses, developers devote more and more time for bug fixing and the time spent on implementing new features decreases. Additionally developers write constructs to make the code more maintainable [3]. Maintenance requires reading and understanding of some familiar or unfamiliar piece of code. And then modify certain sections of the code without causing any further errors. In order to facilitate better understanding, various forms of documentation is done. Further, in the paper we introduce and discuss few tools that allow developers to efficiently embed knowledge with code, which allows to reconstruct the mental model at a later stage.

In order to comprehend the code in totality, developers navigate between various dependencies, class files, methods and data-structures to understand the relationships between various code snippets. Navigating call graphs has been shown to be particularly important

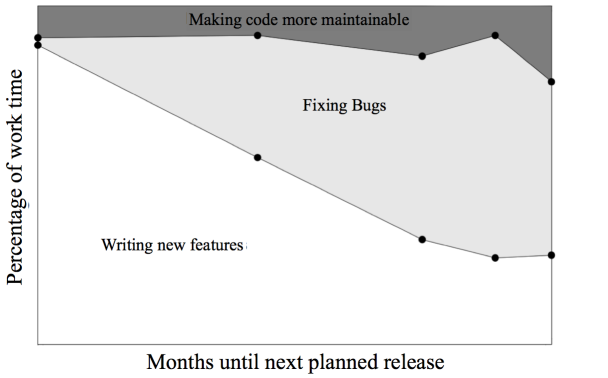

Figure 1: Time distribution per task in a software development project [3]

to understand code and evaluate the locality which could potentially be effected by a modification [4]. IDEs aid developers to navigate dependencies via different methods. For example, project and outline explorers in eclipse. In this paper we will also look at tools, plugins and IDEs that provide efficient and effective means of navigation.Tools that programmers often use have no notion of a task context. We will discuss a few studies that present a models to encapsulate task context and further look at some tools which implement them.

CODE COMPREHENSION

Code comprehension is defined as "a process whereby a software practitioner understands a software artefact using both knowledge of the domain and/or semantic and syntactic knowledge, to build a mental model of its relation to the situation. [5]". Programmers go through material over the internet in order to overcome their cognitive barriers. The related materials browsed over the internet helps a developer to build a current mental model. But this context is lost with code. So, when a developer unfamiliar with the concepts reads the code has to browse over the internet and find valuable material required to comprehend the code. A study conducted at Stanford University shows that developers spend 19% of development time browsing over the internet for examples [21]. In spite of this sting association of developers with examples of the internet, the IDE and the Web Browser are unaware of each other.

Capturing web browser history alongside code edits.

Connecting source code and browsing histories might help programmers maintain context and expedite understanding, when code is shared. HyperSource is an IDE augmentation that associates browsing history with source code edits. Tracking and visualising visited pages

is implement through four components: a browser extension that captures visited Web pages, an augmented editor that can manage captured pages, a source document data structure that maintains associations between code and web pages, and a user interface that enables interaction with this history [7].

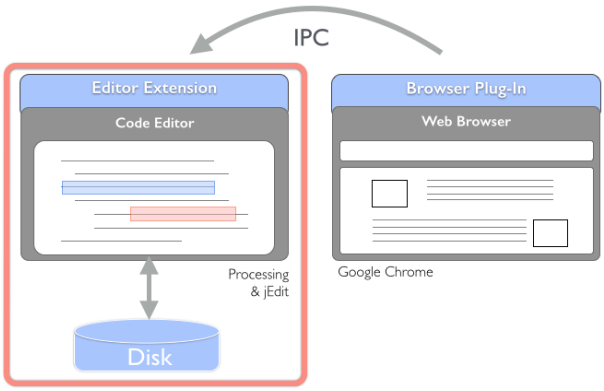

Figure 2: Shows the various components involved in Hypersource. [22]

As the developer loads a new source on the browser, a HyperSource browser extension collect a title, URL and thumbnail of the rendered page. This tuple is then is communicated to the IDE using inter process communication (IPC). This interaction can be modelled as a finite state machine of alternating edit and browse phases. So when a user browses the visited pages are added a queue. Once he switches back to editing in the IDE, this queue gets tagged with source code edits i.e. new characters. Switching back to browsing clears the queue and then filled again. Early prototypes suggested that not all edits that followed the browsing phase were related. To model this HyperSource considers temporal locality of source code edits and only the first n edits are annotated with visited pages queue (see figure 3). User studies have shown n=4 to be the most appropriate value.

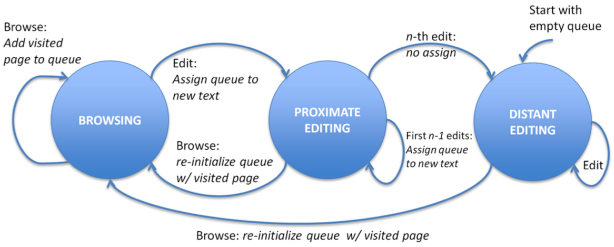

Figure 3: HyperSource modelling edit locality [3].

Not all pages that the developer visits may be relevant. User's interaction with the web page may indicate that it would be particularly relevant. Pages are considered relevant if :

- text was copied from the page into the source document;
- the page was the final page visited before editing,
- the developer marked the page in the browser extension;

Filtering options offered by HyperSource are blacklisting specific domain names, an action to clear internal queue and allowing to remove individual items from the log.

Blocks of example code on the Web are typically situated within a rich context. When the example code is copy pasted from the internet the rich context, explanations and demos are lost. Also, use of these examples necessitates understanding, configuring and integration. Codelets is one such tool which embeds rich environment into the existing code and allows easy manipulation and configuration.

Linking Interactive Documentation and Example Code in the Editor.

Codelets link interactive documentation or helpers with code snippets or examples in the editor. These helpers can be textual explanations of what the example does or, a preview of what it looks like. Codelets can be copied and pasted from a website or can be edited directly inside the editor. Helpers can also be interactive. It can write code for the developer while the developers manipulates a form provided by the helper. It not only allows the user to see what the complete copy pasted code does but also shows localised preview of code. Helpers do not obscure code, they can be hidden and shown when required by the user. (see figure 4). Codelets are implemented *like* web-pages they have *Headers* with meta information about the codelet. Example code is written with *marks,* where the code might change and, helpers are written in *HTML*. Interactivity is implemented through *JavaScript* with an *API* with interacts with the editor. Codelet users can have related explanations attached to their code, which can be easily recalled it if necessary [8].

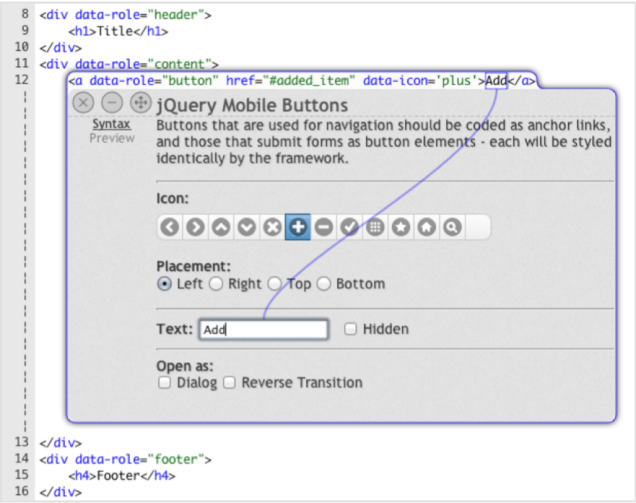

Figure 4 : Shows user interaction with helpers in codelets [8].

A very general practice followed by programmers is to comment their code with textual descriptions of what a specific code snippet does. But, writing good comments is by no means easy. Inexperienced programmers can easily write a verbose comment by talking about everything or make an evasive comment by providing insufficient or irrelevant information. In many situations even experience programmers fail to write self explanatory comments. In such cases, code reviewers find audio or video code narration quite helpful.

MCT (Media Commenting Tool) allows developers to explain the intent of their code by recording video, audio or taking control of the mouse. These comments are embedded with code snippets and can be replayed to comprehend the code in detail when required. [9].

The tools and IDEs introduced till now along with traditional embedded documentation only allows understanding of only a confined locality of code. Overall, software is difficult to comprehend. And, it is necessary to understand it completely before performing a maintenance or a bug fixing task, to ensure that bugs are not introduced any further. In order to do this developers have to navigate through various dependencies of source code.

For example, developers waste a lot of time in navigating dependencies through package explorer, file tabs and text search in Eclipse IDE. Also, while doing so they loose track of the relevant code and they have to repeat the search multiple times causing an overhead. A task context is created by monitoring a developer's activity and extracting the structural relationships of program artefacts [10]. Analysing developer strategies can produce useful insights which can be leveraged by tools to help developers seek, relate and collect information in an effective way.

Modelling Developer Strategies

Two very important studies, one by Andrew J. Ko et. al. (2006) and other by Jonathan Sillito et. al. (2008) analyse developer strategies and present a coherent model. The latter also contributes 44 types of questions that programmers ask during software evolution tasks. These questions are classified into 4 groups which represent the phases in which developers tackle the tasks.

The model presented in 4 phases is as follows :

(a) Finding focus point.

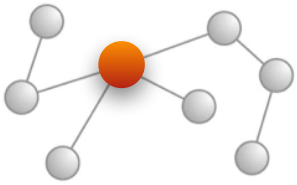

Figure 5a [12]

Here for example, developers ask questions like *"Where in the source code is the error message being thrown?"*. Basically, they try to find a staring point to begin the investigation.

(b) Expanding the focus point.

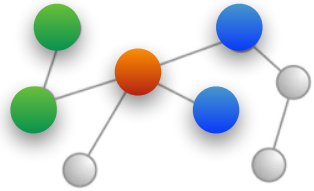

Figure 5b [12]

Here developers expand a given entity which they believe is related to the task by exploring relationships. For example, the questions that may be asked here are: *" Whom does the entity inherit from ?", "Who is calling the particular entity?", "what methods does this entity call?"* etc.

(c) Understanding a subgraph.

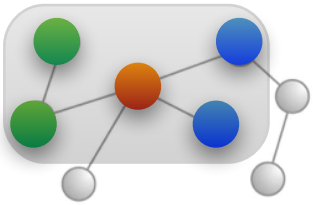

Figure 5c [12]

In this phase, all related entities and relationships form a subgraph. And the goal is to understand the purpose of this graph as a whole. Example questions here could be *"what feature does the subgraph try to achieve as a whole ?", "How are the instance of these types created and assembled ?"* etc.

(d) Questions over groups of subgraph.

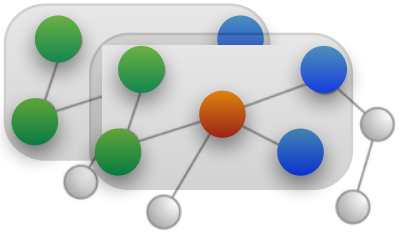

Figure 5d [12]

This phase allows the developer to analyse the relationships between a subgraph and the rest of the application or between multiple subgraphs. For example, the questions which could be asked here are :*"How is the view controller and the model related ?", "How does the control transfer from one subgraph to the other?"* etc [11] [12].

The sequence of occurrence of these phases do not necessarily have to be in order i.e. on the the contrary there can be a drill down strategy deployed by the developer than a panning out strategy. Developers build a task context by going over these phases and navigating the call graph. Call Graph is a graph in which each node is a method and directed edges pointing towards the node of the *callee* method from the *caller* method [18].

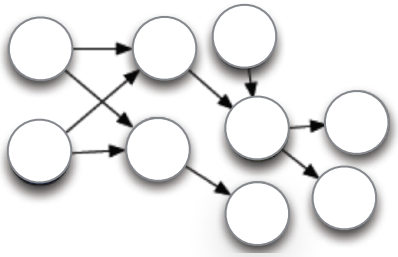

Figure 5e : Call Graph [18]

IDEs like Eclipse, by default do not have a notion of task context, as it is basically a text editor with a file browser. So, the IDE has to encapsulate the task context in order to provide extra aid to developers.

One way to do this is by abstracting away irrelevant information irrelevant to the current task context. Strategies can be applied to gather quantitative information to model developers task context. Different tools use different heuristics to solve this. One such tool is Mylyn.

Mylyn

Mylyn makes use of Degree of Interest (DOI) model to facilitate task oriented programming and debugging. It reduces the information overload and makes multitasking easy. For example, the user can select his required task and the project explorer will automatically filter the irrelevant elements by displaying only the elements required based on the DOI model of the selected task. The DOI model is built based on the Developer's programming activity. Whenever the user selects a programming element its DOI value increases and over time if the element is not selected its DOI value decreases [20].

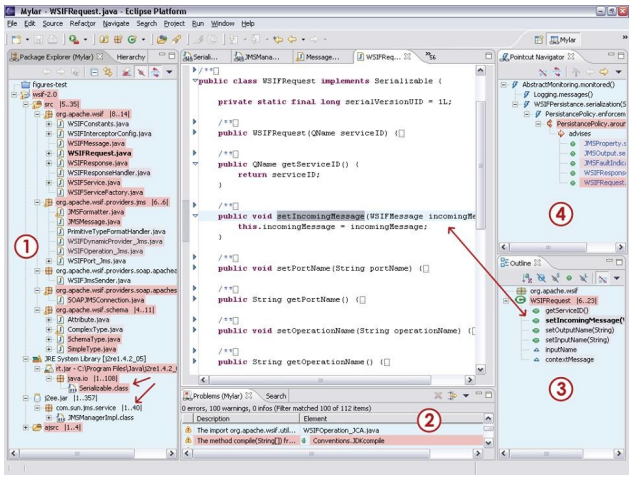

Figure 6 : Mylyn views 1)Mylyn Project Explorer 2) Mylyn Problems list 3) Mylyn Outline 4) Active Pointcut Navigator

Mylyn extends Eclipse Java Development Tools (JDT) and AspectJ Development Tools (AJDT).Mylyn maintains a problems list where the problems related to the DOI are highlighted to stand out from the list. The Mylyn Outline shows only the members related to the particular task. The list can be folded or unfolded to reflect the filtering state of the Mylyn Outline. The Active Pointcut Navigator is capable of showing all of the crosscutting related to that context [20].

Mylyn does not completely change the user interface of Eclipse IDE, but adds a transparent layer over the existing user interface to display only the task oriented information. Figure 6 depicts the user interface of Eclipse IDE with the Mylyn views. A user study was conducted over a period of 5 days to see whether the users prefer Mylyn view or the default view .The study revealed that users prefer Mylyn over the default view [20].

Instead of maintaining the same user interface some IDEs like Code Canvas and Code Bubbles afford task oriented programming by completely changing the code visualisation. Both designs had the same goal is to provide the user all the necessary information needed for a task in a spatially stable way [15].

Changing the Code Visualisation in Integrated Development Environments

Code Canvas and Code Bubbles change the existing "bento box" design (tabbed documents and tools) of IDE by introducing a pan and zoom interface that hosts the software project's code [15].The pan and zoom interface tries to overcome the following drawbacks of the "bento box" design [16].

- Developers get frustrated by the disorientation of the code when they try to find solutions to their questions in the code.
- Developers find it difficult to synthesise the required information as it is displayed in different areas of the screen.

Code Canvas replaces the bento box with a canvas which makes use of semantic zooming. Semantic zooming displays various levels of information detail at various levels of zoom. Code canvas uses MSAGL graph layout to produce the initial code layout. It allows viewing the entire system and its architecture in a smooth "deep zoom" kind of way. Figure 7a shows the user interface of code canvas. The layout can be modified in two different ways [16].

- The user can add containers that represent concepts that are not syntactically explicit in the code.
- The user can split the code file into fragments by tearing off a method or a number of methods. Each fragment has a unique colour to identify its code file.

Code Canvas can enhance visualisation by displaying various layers of information such as search results, test coverage and execution traces .The user is

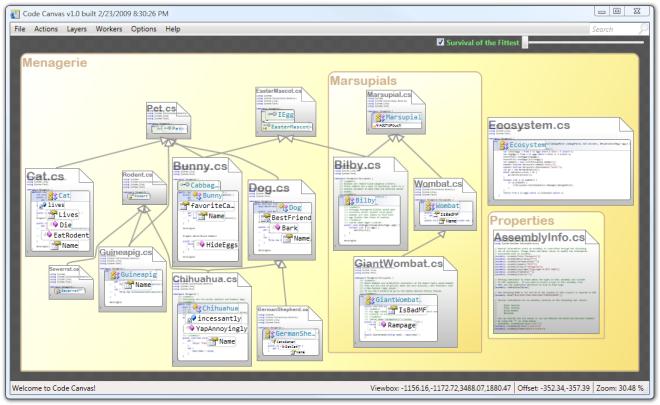

Figure 7a : Code Canvas User Interface

independent to add multiple layers depending on the information required by the user. Layering makes synthesising the information easy. The ordering of the graphics drawn in a layer is done along the Z axis. Code canvas has various layers: directory structure; file

structure; class diagrams; code test coverage; document editors; definition labels; code annotations; execution traces; search results; and reference edges .These layers are listed in Z order from back to front order[16].

Code canvas allows the user to create multiple canvases. Multiple canvases help multitasking and also make performing detailed tasks easy by allowing the user to perform in two distant parts of the canvas simultaneously. Multiple canvases can be created by dragging onto other monitors or can be docked with other canvases in a tabbed browser. Each canvas has its own viewport, its own level of zoom, its own set of active layers and its own filtered set of item. The user can make use of filtered canvas to filter the required items. For example, if the user is making use of call stack layer the filtered canvas will contain the parts involved in that layer [16].

The spatial position of the fragments is maintained because it has a meaning in the context of the overall software diagram and cannot be replaced arbitrarily. Deployment teams can share and collaborate on a single master canvas but in the same way they should also agree on the locations of the files in source control directory tree because each item on the canvas has only one location and is never duplicated. The canvas serves as a map of the project allowing the user to form and exploit spatial memory [16].

Code bubbles also have similar features but instead of denoting parts of the code as fragments, the methods of the code are denoted in bubbles. Along with source fragments it can also contain tool windows, property grids, tear-off queries, work items, email, etc. Task oriented working sets are created by using various types of bubbles like Javadoc bubbles, Note bubbles, Flag bubbles, Web bubbles and Bug bubbles. A "Workspace Bar" provides the necessary definition of the working sets for a particular goal. The items can be duplicated in Code Bubbles. In Code bubbles maintaining the spatial memory is not the primary focus. The position of each item on the canvas is meant to maximise organisation or screen real-estate for the current task. It is also possible for the development teams to make copies of the created canvases [17].

Since Code Canvas and Code Bubbles introduced the same concept at the same time they decided to merge together along with Microsoft Visual Studio Ultimate team to develop Debugger Canvas [15].The main goal of the design was to gain a balance between allowing the users to focus on the task and form an opinion on the bubble design while also allowing the users to remain comfortable with the existing user interface [15].

Debugging made Faster and Easier

In order to strike the balance, Debugger Canvas did not change the user interface completely. Debugger Canvas focuses on debugging as debugging is a cognitively intense and navigation-heavy activity. It made use of canvas only in debugging mode. It is implemented as a plug-in for Microsoft Visual Studio [15].

When debugging is done with a break point, the Debugger canvas will open the canvas containing the focus method where the break occurred in a bubble. Stepping through that code will display the whole call stack with a separate bubble for each method that runs. Each bubble contains a button at the top right corner of it. Clicking that button will display all the local variables in that method. We can also save the current value of the variables in the object's current state and compare these values with the new values of the variable when the objects change state. Alternatively, we can also drag a method into the canvas and generate the whole call tree of that method [15].

The user can also debug concurrent programs using Debugger Canvas. Debugger Canvas will display methods of different threads in different horizontal bands with differently coloured bubble border for each thread. When the same method is executed by different threads, different copies of that method are created in separate bubbles for each and every thread. The currently executing bubble is showed by a thick yellow border. Each bubble has a pop-up containing the local variables and a highlighter bar for its program counter. The highlighter bar of a method advances whenever the method executes due to a thread switch [15]. Figure 7b illustrates this feature.

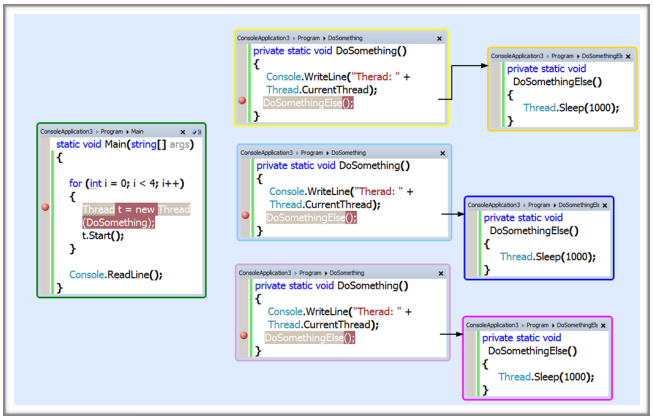

Figure 7b : Debugger Canvas User Interface

Debugger Canvas adapts a non-historical design. The user can duplicate the source code fragments on the canvas each time the code is stepped into, or the same fragment may be used again and again if called many times. It re-uses the same canvas in each session of the debugging unless the user explicitly specifies to open a new canvas [15]. This approach of Debugger Canvas has its advantage and disadvantage. The advantage would be that it would be easy for novice users as it reduces the overhead of a new window management concept .The disadvantage would be that the users will not be able to perform code comparison as performed using Code Bubbles [15]. Developers can send only the screen shots of the canvas, collaboration is not possible. The lifetime of the canvas is only for a single debugging session.

Though this design strikes a balance between the new visualisation and existing user interface, developers will find it very difficult to accept the change. Certain tools such as Stacksplorer and Blaze make task oriented navigation efficient by just augmenting the existing user interface without making major changes to it. Both of these are plugins to the Xcode IDE.

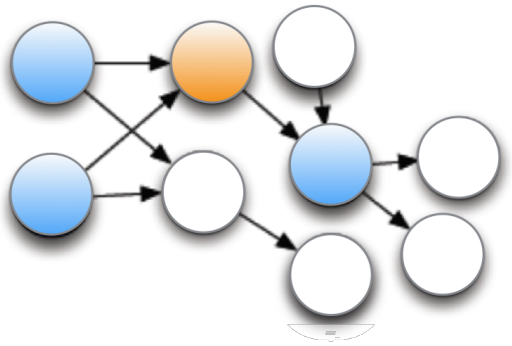

Figure 8a: Call graph visualisation for Stacksplorer [18]

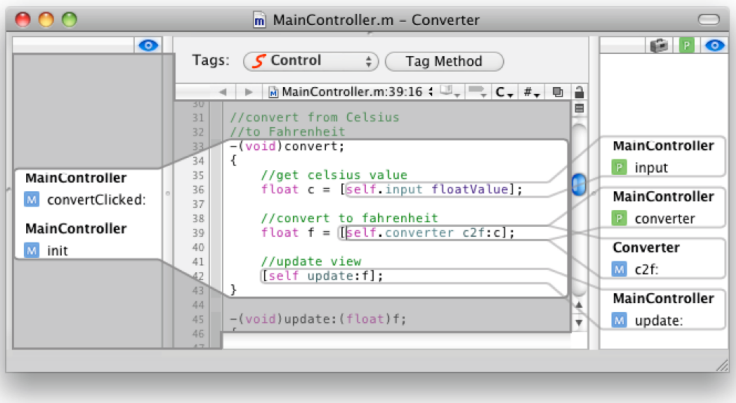

Figure 8b: Stacksplorer user interface [1]

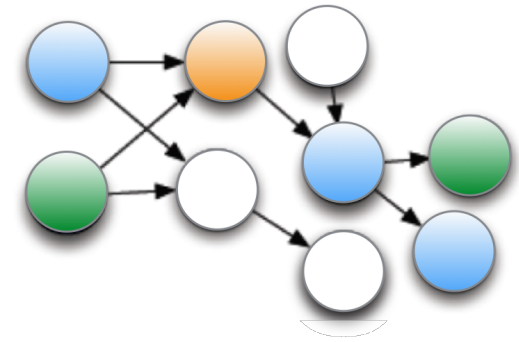

Figure 9a Call graph visualisation for Stacksplorer [18]

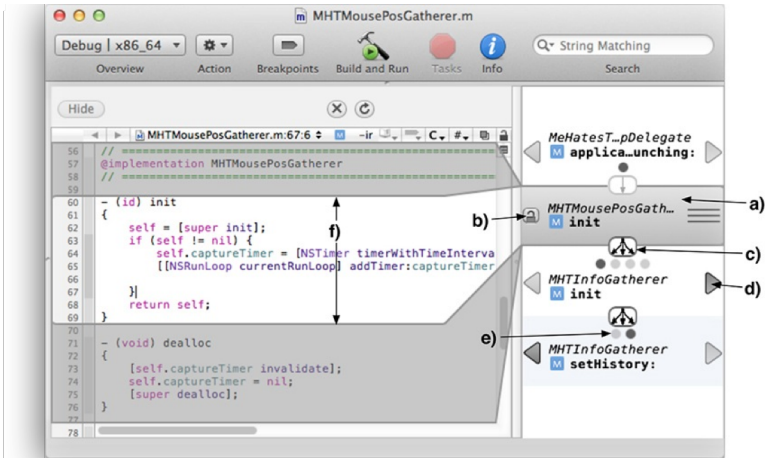

Figure 9b: User Interface of Blaze a) Focus method. b) Toggle Button to lock or unlock the focus method. c) It reveals all options. d) switch to next/previous alternative e)shows the number of options and the current state. f) connect the currently edited method with the corresponding item on the path [19].

Augmenting the Existing User Interface

Focus method is the method that contains the information required for the user or the information necessary to the task performed. Users should navigate code to find the focus method. A formative study was conducted to see how developers navigate in Objective-C using X-Code with respect to Java developers. The study revealed that both navigate similarly.

Stacksplorer makes navigating around the focus method easy by visualising the immediate neighbours in the upstream of a method that is the methods that are calling the focus method and immediate neighbours in the downstream of the method that is the methods that are called by the focus method instead of visualising the entire tree of the focus method as in the case of Call Hierarchy [1]. Figure 8a shows the visualisation of the call graph in Stacksplorer. Each node in the graph represents a method. The light orange colour node is the focus method and the blue coloured nodes represent its neighbourhood [1].

As shown in Figure 8b whenever the user selects a focus method in the central editor, two columns are shown, one in the right and one in the left. The right column shows the list of callees of the focus method and the left column shows the list of callers of the focus method [1].Choosing a method from the left or right column will not change the visualisation with respect to that method. The focus method in the central editor is locked which makes the navigation around it easy in the right and left columns. The focus method can be changed anytime by choosing a method from the left or the right column or a different method directly in the central editor.

Stacksplorer supports assigning tags to a method and bookmarking important paths in a call graph as a form of communication between the methods. Tags reveal the purposes served by the method or which features of the application uses that method. The tags can be named and can be visualised in coloured frames [15]. In some cases it would be necessary to explore a particular path of the focus method in depth, one such tool that facilitates it is Blaze [18].

Two-Phased Call Graph Navigation

The call graph exploration tools explore the methods in two phases. Phase 1 is exploring around the focus point based on the scent and Phase 2 is the direct access to the methods of interest. Blaze is designed to support both these phases. Blaze is used to navigate a particular path of the focus method. Figure 9a shows the visualisation

of the call graph in Blaze. The light orange colour node is the focus method and the blue coloured node gives a single path including the focus method and the green coloured nodes depict the alternative paths [19].

As shown in Figure 9b, all the methods in a particular path are listed in a separate column on the right hand side of the code editor [19]. The column on the right hand side shows a list of methods along with the focus method. The callers of the focus method are above the focus method in the list and the callees are below the focus method in the list [19].

The main difference between the Call Hierarchy tool and Blaze is that the focus method can be locked or unlocked so that navigating through the list does not change the focus method. Maintaining the state is optional which can be toggled with the help of a button [19].

During Phase1, the focus method is unlocked to facilitate the user to find the focus method by navigating the call graph. Navigating to a different method in the central editor will change the list of methods in the right side column with respect to that method. During Phase 2, the focus method can be locked allowing the user to explore all the paths with respect to the locked focus method.

Stating different tools for navigating call graphs is necessary to analyse the performance of these tools by comparing their task completion time [19].

Comparison of Xcode, Call Hierarchy, Stacksplorer and Blaze

A user study was conducted to compare the performance of various code navigation tools. The user study had a between-group experimental setup with 35 subjects. In the study, 9 of the subjects were exposed to Call Hierarchy condition and 8 subjects were exposed to Xcode, Stacksplorer and Blaze conditions. The subjects had to perform a Complete Trail consisting of 2 tasks. The subjects performed these tasks on the source code of Bib Desk [19]. In Task 1, the subjects had to do a particular change to the BibDesk's Auto-file feature. The subjects had to prepend a particular string before the generated file name. In Task 2, the subjects had to find the side effects of the changes they made in Task 1 [19].

Quantitative analysis depicted that many participants completed the trial using code navigation tools than without any tools. Figure 10 depicts the box-plots that shows the task completion time for Task 1, Task 2 and the Complete Trial. For the complete trial, the comparison shows a significant advantage of Stacksplorer and Blaze compared to Xcode and the Call Hierarchy (p=0:001, F(1;25)=12:76) [19].The difference is also significant for Task 1 alone(p = 0:004, F(1; 28) = 9:95).The possible reason for this difference in Task 1 is supposed to be the efficiency of Call Hierarch in solving Task 2 alone .

The difference was not significant between Blaze and Stacksplorer in the Complete Trial (p = 0:882, F (1; 25) = 0:02).The possible reason being the Stacksplorer's efficiency in Task 1 is counterbalanced by Blaze's efficiency in Task 2 [19].

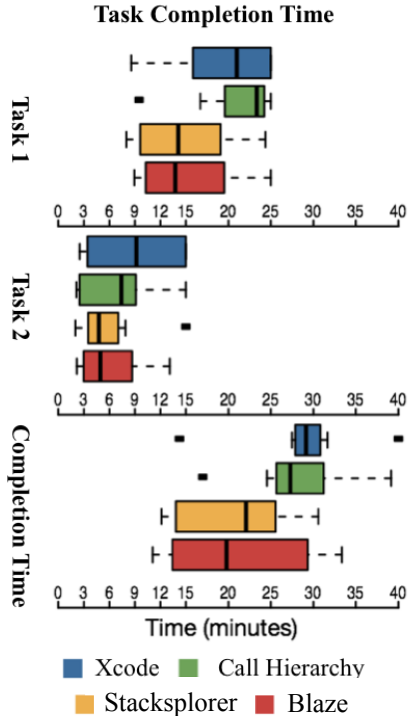

Figure 10: Task completion times by task and condition [19].

Qualitative analysis depicted that the tools were more frequently used when the starting point was clearly given in the description. Lack of automatic updates and the explicit switch between the caller and callee view modes caused mode errors in Call Hierarchy. Blaze and Stacksplorer avoided these problems due to enhanced visualisation. Blaze and Call Hierarchy prevents the user from getting lost from the focus method by keeping an explicit reference to the focus method [19].

Everything discussed till now works with synchronous code, where developers can step through chain of events. Asynchronous code cannot be stepped through by a debugger.

Understanding Asynchronous Code

Theseus is a debugging interface used in order to make sense of control flow and addressing of asynchronous code. It uses the program trace in order to give a realtime in editor feedback which developers can utilise to answer reachability questions [4]. It allows developers to interact with the program's runtime state like a step debugger.

The Theseus interface is available as an extension of Brackets text editor. Theseus starts up when a developer opens a web page from inside of the editor. Now as the webpage gets rendered in the browser, Theseus augments the editor with blue pills. These pills appear next to a function to indicate number of times it has been called. Developers can click on these pill to query. Functions and code events as a part of this query are shown in a small window opened at the bottom of the editor [21]. This shows values of all the variables which were in scope, to be inspected. Figure 11a shows the different UI augmentation of Theseus.

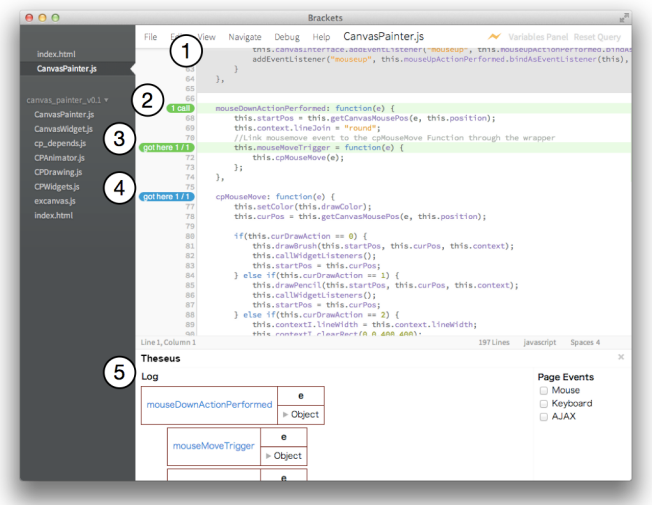

Figure 11a: Basic elements of Theseus interface [21]
1) Code that has yet to be executed is given a Gary background.Pills appear next to functions to display the number of times they have been called.
2) When the user clicks to add them to the query, they turn green.
3) Call counts become relative to the root of the query when one is active.
4) A query extend to every other function
5) Functions and page events that are part of the query show up in this area with the values of variables that were in scope, where they can be inspected.

In the example shown in Figure 11b, the developer writes a code, where a button click requests for search results. But, it does not work. From the Theseus interface it is evident that the *success* event handler does not get called.

Figure 11b: Grey highlighted statements indicate [21]

So, the user queries by clicking the *pill* for the ajax call and looks up the related calls. In the helper window it can be seen that the Ajax request results in a '404 Not Found' Error. The developer realises that this had happened as there was a typo in the URL, i.e. *serch* instead of *search*.

Figure 11c: Theseus helper window [21]

Theseus effectively answers reachability questions concerned with the control flow, which are the most difficult to answer without special tools.

SUMMARY

Understanding code written by other developers has always been difficult. Also recreating the mental model again, reading the code written by oneself is a time consuming task. Tools like Codelets, HyperSource and MCT allow to embed essential knowledge required to create this mental model. To understand software completely it requires developers to navigate through the various dependencies. This task can be made easier by capturing user's task context. Mylyn is one such Eclipse plugin which does this by abstracting away irrelevant information w.r.t the task. Code Bubbles and Code Canvas are canvas based IDEs which allows users to relate source code with spatial information and explore the call hierarchy by stepping through it. Such IDEs allow to arrange different task spatially, in turn allowing the developer to create his task context. Such IDEs face challenge of adaptability i.e. they do not have immediate usability as developers are accustomed to file based environments. But the canvas based IDEs have proven very useful for novice programmers and debugging. Call graph navigation is the key to source code comprehension. Tools like Stacksplorer and Blaze are an advancement over Call Hierarchy. IDEs, plugins and tools can exploit certain aspects of developer's behaviour and help them code effectively and efficiently.

REFERENCES

1. Stacksplorer: Call Graph Navigation Helps Increasing Code Maintenance Efficiency, Jan-Peter Krämer et al. UIST'11.
2. Software Engineering: A Practitioner's Approach. McGraw-Hill, 7th edition, 2010.
3. Maintaining Mental Models: A Study of Developer Work Habits. T. D. LaToza et al. ICSE '06. ACM, 2006.
4. Developers Ask Reachability Questions. T. D. LaToza et al. ICSE '10, pages 185–194. ACM, 2010.
5. Software Comprehension – A Review & Research Direction, Michael P. O'Brien, University of Limerick, Technical Report UL-CSIS-03-3
6. Connecting source code and web resources, Goldman, M. et. al., Journal of Visual Languages & Computing (2009),
7. HyperSource: Bridging the Gap Between Source and Code-Related Web Sites, Björn Hartmann, CHI 2011.
8. Codelets: Linking Interactive Documentation and Example Code in the Editor, Stephen Oney et.al, CHI 2012.
9. MCT: A Tool for Commenting Programs by Multimedia Comments. Yiyang Hao et al. Peking University, China.
10. Using Task Context to Improve Programmer Productivity, Mik Kersten et. al.
11. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during

Software Maintenance Tasks, Andrew J. Ko, ITSE 2006.

12. Asking and Answering Questions during a Programming Change Task, Jonathan Sillito et. al., ITSE 2008.
13. Using Information Scent to Model the Dynamic Foraging Behaviour of Programmers in Maintenance Tasks, Joseph Lawrance et. al., CHI 2008.
14. Reactive Information Foraging: An Empirical Investigation of Theory-Based Recommender Systems for Programmers, David Piorkowski et. al., CHI 2012.
15. Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm,Robert DeLine et. al. ICSE 2012.
16. Code Canvas: Zooming towards Better Development Environments, Robert DeLine et. al. , ICSE 2010.
17. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments, Andrew Bragdon et.al, ICSE 2010.
18. How Tools in IDEs Shape Developers' Navigation Behavior, Jan-Peter Kramer et. al., CHI 2013.
19. Blaze: Supporting Two-phased Call Graph Navigation in Source Code, Jan-Peter Kramer et. al., CHI 2012.
20. Mylar: a degree-of-interest model for IDE, Mik Kersten and Gail C. Murphy, AOSD 2005.
21. Theseus: Understanding Asynchronous Code. Tom Lieber, CHI 2013