

Software Journeys

by

Adrian Filip

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

©Adrian Filip 2012

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Getting familiar with the code is a challenging activity and therefore resource intensive. The larger the software code base, the larger the resource expenditure. We consider software development in the case of established software developed by mid to large mature teams consisting mostly of experienced developers working for mid to large size mature organizations. Any increase in the efficiency of this activity would result in significant competitive advantage for a for-profit software company. This thesis explores a new way of documenting code that could increase the productivity of software development.

The method consists of creating small, dynamically ordered sets of code locations called Landmarks. These sets called Journeys are significant for a feature. The landmarks contain documentation related to system behavior and qualitative system state information at the time when the software execution reaches the locations. A plug-in tool for Eclipse was built in order to capture and utilize this information. Except for this light plug-in, this new type of documentation is very light and does not require additional software systems for management therefore, it adds no configuration management requirements. This information is stored, and shared in a seamless manner via the existing source control systems.

An experiment was performed to gauge the efficiency of this method versus the current development practice. The difference of productivity between developers not using this approach versus developers benefiting from this approach was captured. The results could be qualitatively interpreted as pointing towards an overall increase of productivity for the participant developers using the new approach.

Acknowledgements

I would like to thank my family for support and for encouraging me during different stages of completing this work. I am grateful to my supervisor, Prof. Derek Rayside for being supportive and patient throughout my program, and always demonstrating high professionalism and knowledge. His outstanding help was essential to this work. I would also like to thank Patrick Lam and Reid Holmes for their interest and helpful comments that made me explore more dimensions of this subject.

Dedication

To my family, friends and colleagues that made this possible.

Table of Contents

AUTHOR'S DECLARATION.....	ii
Abstract.....	iii
Acknowledgements.....	iv
Dedication.....	v
Table of Contents.....	vi
List of Figures.....	ix
List of Tables.....	x
Chapter 1 Introduction.....	1
1.1 Overview.....	1
1.2 Motivation.....	2
1.2.1 Motivational Example.....	3
1.3 Thesis Structure.....	5
1.4 Definitions and Terminology.....	5
1.5 Objectives.....	6
Chapter 2 Related Work.....	8
2.1 Research Studies.....	8
2.2 Existing Solutions.....	14
2.2.1 Research Solutions.....	14
2.2.2 Documentation based approaches.....	15
2.2.3 Industry solutions.....	20
2.3 Tools with higher degree of similarity.....	22
2.3.1 JTourBus (Oezbek , Prechelt).....	23
2.3.2 TagSea (Storey, Cheng et al.).....	23
2.3.3 Tours or Enhanced Routes for TagSEA (Cheng, Desmond et al.).....	24
2.3.4 Pollicino (Guzzi, Hattori et al.).....	25
2.3.5 Feat (Robillard).....	25
2.3.6 ConcernMapper (Robillard, Weigand-Warr).....	26
2.3.7 Detailed comparison of Software Journeys with other tool based solutions.....	26
Chapter 3 Perspective and Novelty.....	29
Chapter 4 Solution.....	30
4.1 Usage Scenarios.....	30

4.2 Problem description	33
4.3 Scope.....	34
4.3.1 Core scope.....	34
4.3.2 Requirements	36
4.4 Solution.....	37
4.5 Analysis.....	39
4.5.1 Strategies for Associating Journeys with existing SDLC concepts	39
4.5.2 Grouping	41
4.5.3 Selection Granularity	42
4.5.4 Training.....	42
4.5.5 Information Volume.....	43
4.5.6 Documentation Location.....	43
4.5.7 Documentation and life-time of software	44
4.5.8 Non-runnable code resources.....	44
4.5.9 Extending OOP concepts to documentation	45
4.5.10 Optional features	47
4.5.11 Alternate Journeys	50
4.5.12 Synchronization	51
4.5.13 Extensibility	53
4.5.14 Navigation.....	53
4.6 Design	54
4.7 Design principles	54
Design to optimize the use of existing resources	54
Design for usability.....	54
Design for clarity and minimal additions to existing user-interface	54
Design for flexibility and extensibility	54
Use an incremental approach to minimize user confusion.....	54
Use predictable behavior.....	54
Design for robustness.....	54
Design for adoption.....	54
4.8 Design Decisions	55
4.8.1 Creation of Landmarks	55

4.8.2 Views	56
4.8.3 IDE context	57
4.8.4 Customizations of Journey Executions	57
4.8.5 Blending Journeys.....	57
4.8.6 Landmark granulation level	58
4.8.7 Landmark visitation constraints	59
4.9 Implementation	60
Chapter 5 Experiment	61
5.1 Experiment Selection	61
5.2 Sample Software used by the Experiment	61
5.3 Documentation used by the Experiment	62
5.4 Participants.....	63
5.5 Preliminary Survey	66
5.6 Results.....	67
5.7 Experiment Set-up	71
5.7.1 Experiment specific instructions for group P (Paper).....	73
5.7.2 Experiment specific instructions for group J (Journey).....	73
5.7.3 General information about the sample software	74
5.7.4 Tasks	75
Chapter 6 Discussion	78
Chapter 7 Conclusion.....	82
7.1 Summary of Contributions.....	82
7.2 Future Directions	82
Bibliography	84

List of Figures

Figure 1: CodeMap rendering example	19
Figure 2: 3D Visualization rendering.....	20
Figure 3: Example of typical JavaDocs documentation in JHotDraw	21
Figure 4: Example of JavaDocs Documentation.....	22
Figure 5: Late binding of landmark documentation achieved by migrating parent method documentation to child method node.....	46
Figure 6: Landmark properties.....	50
Figure 7: Creation of Landmarks from the main menu and decorator bar.....	55
Figure 8: Creation of Landmarks from the main toolbar and Journeys view	56
Figure 9: Creating landmarks for blocks of code.....	59
Figure 10: Restricting the number of visits for a landmark.	60
Figure 11: Example of class level documentation that was not added to Journeys	63
Figure 12: Experiment detailed results	69
Figure 13: Median time for task 1 to 4.....	71
Figure 14: Example of JHotDraw feature Journeys.....	74
Figure 15: Task 1	75
Figure 16:Task 2	76
Figure 17: Task 3	76

List of Tables

Table 1: Detailed comparisons for first set of characteristics	27
Table 2: Detailed comparisons for second set of characteristics	28
Table 3: Participant experience.....	65
Table 4: Survey questions.....	66
Table 5: Survey answers	67
Table 6: Raw Data	68

Chapter 1

Introduction

1.1 Overview

Management of mid to large-size code base projects developed by mid to large teams requires efficient use of resources. The process is led by development managers that establish and implement software development processes. One of the responsibilities of the development managers is to adapt and optimize these processes for maximum efficiency.

During the software development process, a significant amount of effort is taken by activities focused at understanding unfamiliar code, or code that was once familiar but the information faded in time. Developers not familiar with the code usually take longer to implement new enhancements and often produce a lower quality code.

In a mid to large code-base system, developers can certainly not remember all the lines of code and their associated information but just some locations that are more significant in the implementation of a software feature.

The information exists at the point of code development, however, it is not usually captured while still fresh in the memory of the developer. We call these significant code locations Landmarks. We could look at a feature as a Journey the program execution takes through significant code locations.

It is important to capture this information and offer tooling for other developers to re-live the journey.

This is a new form of documentation that connects features/work-item-based documentation with code. It is not an investigative type of approach aimed to recreate lost information using detective style approaches. This thesis investigates and provides a solution for capturing software journeys and landmark information by original developers. It also offers a solution for a developer unfamiliar with a certain feature/use-case to embark and re-experience a software journey therefore acquiring essential information about the code. The experiment considers the return on investment as a result of capturing this information. The increase in efficiency in subsequent use of this information should demonstrate the value of introducing this as part of the software development process.

The focus is on experienced teams working on mid to large code base software and following a mandatory development process with a certain level of formality. It does not cover small or junior teams, developers working in isolation, or software development of small applications.

1.2 Motivation

During my activity in the software development field, I have experienced first hand the need to understand code written by other developers and I discovered that to be one of the most challenging activities. While the challenge is considerable for small systems, it increases dramatically with the size of the code base and the 'age' of the software.

Not understanding previously written code leads to defects, low or no reusability, inconsistent styles, lack of productivity and ultimately missed deadlines and poor software quality.

Later in my career, while managing development teams I realized the lack of support developers experience in this critical area.

This lack of support comes from tooling, process, and management.

It applies both to code change for developing new features and to defect fixing activities.

According to (Latoza, Venolia et al. 2006) *"Developers reported spending nearly half of their time fixing bugs"* so both types of activities are equally important.

A loss of information during development is common in the industry and I observed it to be consistent even for strictly managed projects. A Microsoft study, (Latoza, Venolia et al. 2006) states: *"In the interviews, design documents were described almost as write-only media, serving to structure the developer's thinking and as an artifact to design-review, but seldom read later and almost never kept up-to-date. Another survey talked about 51% of the design documents being kept them up-to-date."*

While this appears troubling at the first glance, it is perfectly justified because the cost of capturing and managing information often surpasses its benefits. Software development is governed by risk management.

The risk of spending resources in capturing information that might not offset the investment is high, and no experienced leader would consider it acceptable in a business oriented endeavor.

That being said, I always thought that in most cases the loss of information required for developers to get familiar with the code is a wasteful loss because of the high cost of code comprehension. The waste would be more substantial in cases corresponding to large codebase software, mature software and midsize to large teams, producing business software. The (Latoza, Venolia et al. 2006) study concludes: *“Developers spend vast amounts of time gathering precious, demonstrably useful information, but rarely record it for future developers”*

During my career, I practiced and observed ‘classic’ techniques employed by developers to rediscover lost information that leads to familiarity with the code. One of the most common techniques is running the software in debug mode and stopping in several key locations stepping through relevant code. The Microsoft study (Latoza, Venolia et al. 2006), one of the very few that employed experienced developers concluded that developers use this about 28% (subject to a margin of error) of the time dedicated to comprehension using this technique.

While that is quite prevalent, no software support for that activity was provided. In several crucial occasions, when full teams of developers were moved on new projects I observed months of team efforts dedicated to this goal. It was my conviction that those expenditures could have been largely avoided using a multifaceted approach. That would have resulted in a significant competitive advantage.

This envisioned multifaceted approach consists of:

- Creating a new type of documentation linking features to code
- Creating tooling for capturing and consuming this documentation
- Embed the procedure in the software development process
- Manage the implementation of the process

1.2.1 Motivational Example

As a result of a quarterly meeting, a need of experienced development team members to augment a team developing Insurance software was identified.

Another topic, focused on the fourth version of a document-management software developed in North America. This software, reached a level of high level of stability characterized by a reduced number of defects and requests from the customers. It was decided that a document management software could be

reassigned to a low cost based development location and that would result in significant savings. The team developing the document management software was mature, used Java and Eclipse so it was an excellent fit for the existing Insurance Software Needs.

The decision was made and a plan to train the low cost base location team on the document management software was set.

All the existing documentation was rapidly transferred. The documentation was quite significant and contained general architecture documentation, use cases, test cases, high-level design documentation, and detailed level design documentation. While the high-level artifacts use cases, and test cases were mostly up-to-date, the code did not contain significant documentation at the detail level and the existing one was out of date therefore overall unreliable. The detailed documentation was created before coding to support the coding phase, so it did not contain all the details and sometimes the coder took other approaches, mainly to deal with performance issues.

A two-week in-class training program was devised and several developer resources were deemed trainers and flown to the new location. The same scenario took place on the other reassignment. In both cases, the resources that originally developed the software became virtually unavailable due to their own development tasks.

As a result, after the two weeks of training elapsed most developers were assigned new features. Since the code base for both systems was very large a predictable state of insecurity ensued because developers not familiar with the code needed to make quality and timely changes to the code.

The initial approach that seemed to work was executing use cases and adding print statements to the code at guesstimated locations. This worked quite slowly for minor features that traversed a small number of lines.

For larger features, this method was replaced by another method where developers added breakpoints in locations chosen using the same principle. This approach had the advantage that the developer could visualize better the system state including variables and call stacks.

Both methods were based on trial and error and the progress was very slow.

The first batch of new features was late and highly unstable. It took several months until the teams became familiar with the new code.

Had the developers had better tools for their discovery approach this effort would have been reduced.

Had this information had been captured before the software transfer most of this effort would have been unnecessary.

1.3 Thesis Structure

- Brief introduction of this field of research
- Highlight some common traditional approaches in this area.
- Narrow down the field to only several proposed solutions that are the most closely related to this work

- Introduce a solution
- Present solution method, design, and realization
- The experiment
- Discussion
- Summarized results of the contributions
- Future directions

1.4 Definitions and Terminology

SDLC: Software Development Life Cycle

OOP: Object Oriented Development

RUP: Rational Unified Process – one of the standard development process frameworks suitable for OOP development

Beacon: recognizable, familiar features in the code that act as cues to the presence of certain structures. See (Brooks 1983).

Landmark: A concept associated with a code location and specific execution documentation

Journey: A collection of Landmarks created during the development of a feature

Concern: A grouping of locations and other data based on some criteria

Line-item: Usual IBM nomenclature for a project plan task

Work-item: a project plan task

CMS: content-management-system

Unit of work: project plan task

Feature: specific unit of functionality implementation usually corresponding to a work item

Use-case: a standard way of describing functional requirements

Pd : a person day worth of an effort

MVC: model View Controller: a commonly used design pattern

KLOC: code measurement unit for a thousand lines of code

IDE: Integrated development environment

Eclipse: Java development IDE

GIS: Geographic Information System

Mature Organization: A software organization that possesses an organization-wide ability for managing software development and maintenance processes (Paulk, Curtis et al. 1993).

Mature team: a team part of a mature organization that is composed of no more than 15% junior resources.

Defined Capability Maturity Model: *“The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software”* (Paulk, Curtis et al. 1993).

1.5 Objectives

Software Systems with hundreds of thousands of lines of code and sometimes with millions of lines of code need to be changed continuously. The change means adding new or extending functionality, and defect fixing activities.

All this translates into the fundamental activity of code changing. While it is important to consider other development supporting activities, it is the code that provides the functionality the client requires and

therefore is at the heart of software development. In order to change the code the developer has to identify the exact line(s) of code where the change needs to be applied. Changing code is a crucial activity performed by developers. For most of the reasonably managed development, it is the largest expenditure in terms of resources.

The objective is to envision, design, document an approach aimed at increasing efficiency.

A new method of documenting code would be proposed as part of this approach.

In order to study and validate the approach, an experiment aimed at performance measuring needs to be devised and executed.

While it is difficult and highly subjective to measure the quality of coding, a secondary objective is to attempt to gauge the quality of the produced code

Perform the experiment using experienced developers

A subsequent objective is to survey and gauge the opinion of experienced developers relative to this new approach.

Another objective is to design and produce tooling to implement this approach.

Normally for medium to large-scale systems the development is governed by a software development process, we also need to identify at what stages and how this method be incorporated in an existing process.

Another objective is to consider this approach in relation to other vital software development aspects.

Chapter 2

Related Work

2.1 Research Studies

Software comprehension is a well-recognized area and, due to the importance and the complexity of code comprehension, a large number of resources are dedicated to improve its efficiency.

In (Olsem 1995) Olsem calculates that *“Maintenance costs (including personnel and hardware/software usage fees) run as high as 50-80% of the software life cycle resources or \$30 billion a year in the United States alone. Approximately 50% of maintenance costs is spent on just understanding what the software does.”*

Similarly, according to (Ko, Myers et al. 2006) states: *“Much of software developers’ time is spent understanding unfamiliar code.”*

According to (Corbi 1989) *“In a typical system, more than half of the developer's effort is spent on reading and analyzing the source code to understand the system's logic and behavior”*

Despite these efforts, it is clear that no approach in itself can fully address the needs. A multitude of approaches is usually applied to narrow down the relevant code location(s) and essential information associated with those locations. There are numerous research papers and tooling that could be useful to this work. This domain is so important that research started decades ago.

The existing research work supports our solution and helps us in making decisions. The research literature provides clear hints about high-level requirements and sometimes even significant detail.

According to the (Corbi 1989) Program Understanding Project at IBM's Research Division, work began in late 1986 on tools which could help programmers in two key areas: static analysis (reading the code)

and dynamic analysis (running the code). Static analysis work usually preceded dynamic approaches. Our approach belongs in the second area.

In (Latoza, Venolia et al. 2006) the authors state that *“Developers go to great lengths to create and maintain rich mental models of code that are rarely permanently recorded.”* Clearly, this suggests capturing this information.

The limitation of approaches to documentation is probably better described in (Kaelbling 1988) *“surely is a better way to specify location-dependent information than scattering undirected one-dimensional strings through a file”*.

The idea of capturing the information before is lost is highlighted by a number of papers. For instance the (Latoza, Venolia et al. 2006) study states that *“the information tends to remain in the developers’ heads, where it is subject to institutional memory loss”*.

As expected, one of the first steps towards code comprehension is to use the existing artifacts created prior, during and after the code creation. Captured requirements, architecture documents, high level and detail level design documents, test cases, developer guides, coding standards and numerous other resources are used in this process.

The need to use the dynamic system approach rather than a static system approach is argued by (Stroulia, Systva Tarja 2002) research that observes that software systems can only be properly understood at runtime. In their view, that is especially important if the system is an object oriented system based on inheritance and executed methods are selected dynamically based on object type.

Another important aspect underlined in the literature is connecting the information with a specific problem domain. Brooks in (Brooks 1983) states that *“is as important to document the problem domain as it is to document programming concepts”*

The utilization of such resources is obviously dependent on their existence, the degree of detail and most importantly their degree of accuracy. With the exception of in-code documentation, none of these resources is directly linked to code locations and that causes problems. Existing research documents and explains this behavior. In (Latoza, Venolia et al. 2006) paper the authors state that:

“When the code itself does not give the answers the developer needs, one might expect them to turn next to the vast amount of information that is written about it – the bug reports, the specs, the design documents, the emails, etc. This is emphatically not the case. Several factors combined to dissuade most developers from using design documents for understanding code. First, finding design documents was frequently difficult. Design documents were stored on internal websites without a usable search facility, forcing developers to manually navigate hierarchic collections looking for the appropriate design document.”

Ultimately, the code embodies the behavior of the software and is undeniably the most credible source of information.

One might surmise, based on the observations of their behavior published in the literature (Storey 2011) that developers approach their tasks from a risk management mindset.

What would be the point of investing significant and precious time in searching non-code artifacts if at best only 51% would pay off while the rest of the time will result in a loss of investment and probably missed milestones?

We personally believe that the root cause of this is the lack of trust in other non-code form of documentation.

Trust is very important in a development process. Any artifact or tools that fail to project trust with the development group would most likely not be used.

A study done by Parnin and Orso (Parnin, Orso 2011) concludes: *"Developers were quick to disregard the tool if they felt they could not trust the results"*

All our participants in the experiment confirmed their development process did not mandate or monitor software artifact updates after code creation.

From experience, we believe this not necessarily oversight (as this state of facts is common knowledge) but rather a result of tight deadlines and limited resources. There is a cost to pay for maintaining the written documentation. Maintaining formatted documents while doing code development will require switching between different software and production paradigms and introduce disruption. Scheduling updates after the code creation would not only require the effort for the update but also a recollection effort.

The authors of (Latoza, Venolia et al. 2006) conclude that *"Understanding the rationale behind code is the biggest problem for developers. When trying to understand a piece of code, developers turn first to the code itself..."*

Ultimately, this consists of specific code location it is the code-base where developers need to create new code or modify existing code. In medium to large code-base systems this activity becomes similar to the proverbial finding the needle in the haystack.

Probably the most interesting fact is that some hints are constantly pointed out by developers.

The use of launching programs and stopping at different locations has been highlighted by several studies:

During (Sillito, Murphy et al. 2006) experiments *"the participants set breakpoints and ran the application to answer the question "is this the thing" ... This process was repeated until several relevant entities had been discovered."*

In (Wiedenbeck, Scholtz 1989), Wiedenbeck and Scholtz found that developers insert debug statements to understand a program's execution.

The authors of (Sillito, Murphy et al. 2006) observe that *“the debugger was used to help answer questions of relevancy. Participants set break points in candidate locations (without necessarily first looking closely at the code) and running the application in debug mode to see which, if any, of those break points were encountered during the execution of a given feature. If none were encountered, this process was repeated with new candidate points”*

The same finding is presented in (Latoza, Garlan et al. 2007) where during an experiment participants *“also tried out the behavior they were to change by setting a breakpoint and verifying that it was hit”*

The (Sillito, Murphy et al. 2006) study surveys developers that offers solutions like *“We could trace through how it does it's work” and answers “discovered by looking at the call stack in a debugger.”*

The same study (Sillito, Murphy et al. 2006) points to a possible greater quality of the findings using live executions of the software. According to that *“At other times the participants appeared to choose a tool or an approach that was not optimal. For example, ... the participants spent several minutes reading code... This was time consuming and by the end of their exploration, they were left with incorrect information. If they had used Eclipse's debugger (which was available to them) they could have very quickly and accurately answered the question.”*

Another paper, (Parnin, Orso 2011) points out the advantages of using traditional debug tools versus many newer methods: *“When using ... tools, instead of working with the familiar and reliable step-by-step approach of a traditional debugger, developers are currently presented with a set of apparently disconnected statements and no additional support.”*

The idea of connecting work items with Journey/Landmark information (comments) to existing software artifacts is supported by previous research. The (Storey, Ryall et al. 2008) paper states: “*Our research with TagSEA indicated that early adopters are using it to support task management*”. TagSEA is a tool that “*allows developers to tag related code by adding keywords within comments*”

Despite a variety of methods and artifacts aimed at software comprehension is usually produced in a mature development environment a large body of literature points out the fact that developers usually use the code as a primary and most significant resource for comprehension. A comprehensive study done in (Latoza, Venolia et al. 2006) finds that: “*Developers avoid using explicit, written repositories of code-related knowledge in design documents or email when possible, preferring to explore the code directly and, when that fails, talk with their teammates.*”

Documenting medium to large codebases using our method has similarities with the usage of plain comments. As plain comments support rest-of-the-line selection or block selection we needed to understand what approach is more useful. The paper (Ko, Myers et al. 2006) discusses how developers select relevant information and what the scope for that type of information is.

One of the issues is that useful code comprehension information is lost. The literature mentions the significant effort required to acquire familiarity with the code and the fact that information is not captured. The (Latoza, Venolia et al. 2006) study finds that “*Developers spend vast amounts of time gathering precious, demonstrably useful information, but rarely record it for future developers*”

For instance: (Latoza, Venolia et al. 2006) helps us in deciding about the place where the location selection is available, The (Ko, Myers et al. 2006) helps establishing the landmark scope and granulation.

Some of research approaches like (Robillard, Murphy 2007) blend a manual and automatic method. Some require a starting location as a precondition. The same (Robillard, Murphy 2007) paper describes *“The Importance of a Good Seed”* and they *“assumed that a developer knows a relevant program location from which investigation can start. Based on such a starting point (or seed), a developer can investigate related elements in the source code and build a concern graph ... The identification of a seed is a separate phase of a program maintenance task, performed outside of the FEAT tool”*

In (Wilde, Casey 1996) Wilde and Casey referred to this concept as *“places to start looking”*.

The (Sillito, Murphy et al. 2006) study noted that *“The participants in the first study naturally began a session knowing little or nothing about the code and often they were interested in finding any starting point”*

We feel that new tools should relate to existing developer’s knowledge, and be flexible enough to *accommodate sophisticated personal preferences and mental models. Presenting information in a familiar, yet highly flexible context was our goal. There is research work highlighting the pitfalls of contrary approaches. The (Parnin, Orso 2011) paper states that “Unlike traditional debuggers or program slicers, most ranking based automated debugging techniques remove any source of coherence by mixing statements in a fashion that has no mental model to which the developer can relate”*

2.2 Existing Solutions

2.2.1 Research Solutions

The existing research could be studied as part of two large categories. The first category consists of formative literature, which studies this area by looking at the needs and current developer practices. This type of literature usually employs surveys, observations, and developer interviews.

The second category that would be addressed later is evaluative, and formulates solutions, and in many cases performs associated experiments to validate the envisioned solutions. Our work would largely fit in the second category because it proposes a practical solution. While developer interviews are performed, they center on improving the solution.

Studying the existing SDLC artifacts is helpful, and more or less information could be derived, depending on the type of artifact. Some of these artifacts, like requirements specifications are not targeting code comprehension, however, they offer useful information. Some other artifacts like products of architectural nature are more useful to understand the big picture and their value should never be underestimated.

As most of these artifacts don't have a strong connection to code locations, they are in many cases incomplete, obsolete, or even misleading. A research paper (Latoza, Venolia et al. 2006) describes design documents as seldom up-to-date thus unreliable.

Among these solutions, the most prevalent are discovery-based solutions where the information is re-established based on different investigation techniques. Other types of solutions are documentation-based solutions that capture the information for later utilization.

Our solution is a *documentation-based* solution that involves an information collection stage and a consumption phase.

Any of these types of approaches can use query-based approaches, trace based approaches, team-communication based approaches or other methods.

2.2.2 Documentation based approaches

The idea behind the documentation approaches are that the person developing code understands that code at the very detailed level, thus is very familiar with the code and the most authoritative resource in that respect. As the person doing the change is a developer and this information is ultimately geared to another developer, there is no need to transform this documentation in another format or for a different type of audience.

As this information is of ephemeral nature, the task of capturing it should be performed while coding. There are two classic types of documentation: Documentation associated with specific lines of code and documentation associated with code metadata.

This first type of documentation commonly referred to as software comments, exists in virtually all programming languages. This documentation is free form and there is very little standardization in terms of its usage. It is usually used to explain why something is done and in some cases, it is used to explain how it is done. This type of documentation is normally used to add information for uncommon behavior or to break down a larger structure into special purpose sections.

The second type of documentation documents functionality at the static metadata level. For OOP, this documentation is usually associated with methods, class variables and classes. One of the most common forms of this type of documentation is JavaDoc (Oracle). JTourBus (Oezbek , Prechelt) would also be a good example.

None of these forms of documentation is required to be associated with a specific work-item (feature), use case or test case. As the metadata is a static structure so is the associated documentation.

Both these types of documentation add the information directly in the source code therefore the information is available to the whole development team via the source control repositories.

2.2.2.1 *Re-discovery based approaches*

One method to speed up the code comprehension is employing different investigation techniques. Most of the efforts concentrate on extracting that information directly from the code. These approaches are detective-like approaches where information is sometimes reverse-engineered from code artifacts. Some prominent examples are the “*Multi-perspective software visualization environment – ShriMP*” (Wu, Storey 2000) and the “*Rigi environment*” (Kienle, Muller 2010). Another example of this approach is (Greevy, Lanza et al. 2006)

Most of these discovery methods exploit one of more forms of relationship between code module and higher-level entities derived from segments of code.

2.2.2.2 Query based approaches

A common discovery type approach is based on queries. As the name implies, this approach requires the developer to formulate a query that filters and use the results for comprehension. One example of such approach is (Ying, Tarr 2007). Another refined technique is described in (Liu, Lethbridge 2002)

The simplest forms of queries are string searches and this facility is virtually present in all development tools. A mature team with good coding standards would use descriptive identifiers (beacons) thus a query searching for those identifiers would reveal code location candidates.

Some of the approaches query directly the metadata or sometimes other types of relationships like for instance “is a”, “has a”, “uses” etc.

Some of these approaches query for information contained in previously saved execution traces.

Historically, the static type relationships are predominant in this category.

2.2.2.3 Visualization techniques

Once the relevant information has been identified, it has to be presented to the developer.

Most methods display information directly in the code or in metadata explorers, which visualize tree and graph-like structures.

Displaying information in a concentrated graphic format is an appealing proposition and that resulted in a number of solutions that use map like paradigms.

Work in data visualization area by using a dependency matrix is done in (Sangal, Jordan et al. 2005). This work looks at dependencies between the code artifacts to study the coupling and quality of the system.

In (Loretan 2011) and subsequent and related papers and tools like Codemap (Erni 2010), a research group from Switzerland is using maps for code comprehension. This approach produces a graphical representation for the whole system.

These papers use the analogy with the cartography domain and simple cartographic principles are transferred into the software realm.

The resulting tools produce map like outputs that are similar to topographic maps or terrain maps.

In these maps, entities like files are chosen as points on the maps and their size is used to calculate the height of the topographic features. For instance, a large file would be a mountain and small sizes file just a pixel. Much of the effort of the research is dedicated to the best placement of files and the calculations of distance between the maps.

The distance is calculated based on vocabulary similarity, so for instance two files that contain a larger number of words would be closer together. The paper puts most of the effort into the know-how of drawing the graphs and in optimizing the distance calculation process.

The latest tool in the series, CodeMap can extract information from several other Eclipse embedded modules and display it. The most pertinent in this case is 'Call Hierarchy' that can show segments of a journey. The tool shows only atomic segments and also forces showing all possible segments regarding if the segments are part of a journey or not. A subsequent paper discussing the same tool found that software developers were confused by the information on the resulted maps. An example of such map is shown in Figure 1: CodeMap rendering example.



Figure 1: CodeMap rendering example

The Map keyword used by the Software Cartography approach suggest close relation with the Journeys and Landmarks keywords however this association is misleading. There are significant differences between that and our approach. We are not interested in a live journey not a map and we work only from a work-item-oriented perspective. That allows us to filter locations therefore dramatically reduce complexity. We are interested in the journey though the code rather than creating static maps.

This follows a static type relationships analysis path.

Other visualization solutions like (Fronk, Bruckhoff et al. 2006) use graphs or networks to show the components. For instance Figure 2: 3D Visualization rendering: visualizes the core.runtime package of Eclipse.

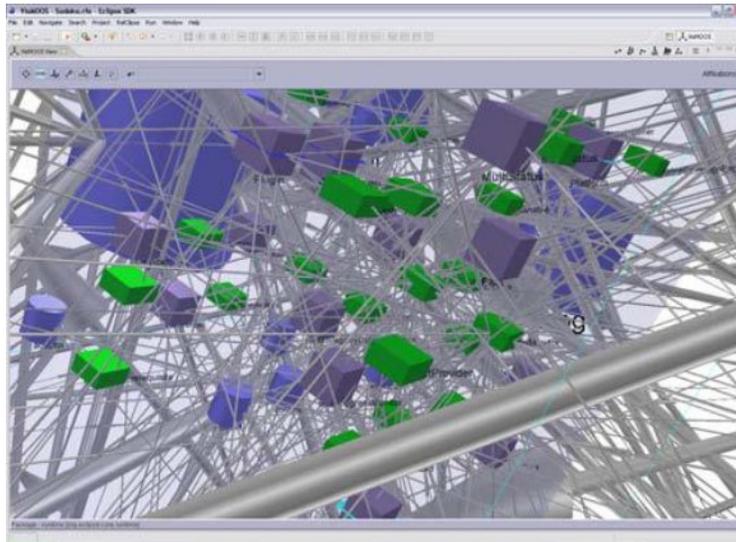


Figure 2: 3D Visualization rendering

2.2.2.4 Team/Communication based approaches

Team communication approaches are based on the idea that information related to code is still present in the memory of the developer that wrote the code. This information can be requested and used by other team members that implement features similar to existing features, extend existing features or fix defects.

Most of these approaches rely on face-to-face communication and some on generic social media communication tools. One of the papers that also study this aspect is (Latoza, Venolia et al. 2006). Obviously, the success of this approach depends on availability of the developers that originally developed the code and the recollection of the information.

2.2.2.5 Other approaches and variants

Historically, the static type relationships are predominant in all the existing approaches. Some solutions like (DeLine, Khella et al. 2005) log source navigation interaction and later mine for high wear to determine places in code.

2.2.3 Industry solutions

One of the most influential developments in the Java space was the creation of JavaDocs.

(Oracle) It pioneered a new type of documentation that is used for writing API specifications and programming guide documentation. This type of documentation was inspired by, “*Literate Programming*” (Knuth 1984)

It introduces a new very simple syntax to documentation. See Figure 3: Example of typical JavaDocs documentation in JHotDraw for an example of JavaDocs comments:

```
/**
 * Deactivates the tool. This method is called whenever the user switches to
 * another tool. Use this method to do some clean-up when the tool is
 * switched. Subclassers should always call super.deactivate. An inactive
 * tool should never be deactivated
 */
public void deactivate() {
    if (isActive()) {
        if (getActiveView() != null) {
            getActiveView().setCursor(
                new AWTCursor(java.awt.Cursor.DEFAULT_CURSOR));
        }
        getEventDispatcher().fireToolDeactivatedEvent();
    }
}
```

Figure 3: Example of typical JavaDocs documentation in JHotDraw

Despite its simplicity or maybe because of its simplicity it had a dramatic impact on the amount of documentation produced for the systems. Mature organizations like IBM made JavaDOcs creation as mandatory part of the development process.

Systems that used to be uncommented or very sporadically commented acquired significant amount of JavaDocs documentation.

The JavaSocs can also be externalized in an external HTML format as in Figure 4: Example of JavaDocs Documentation.

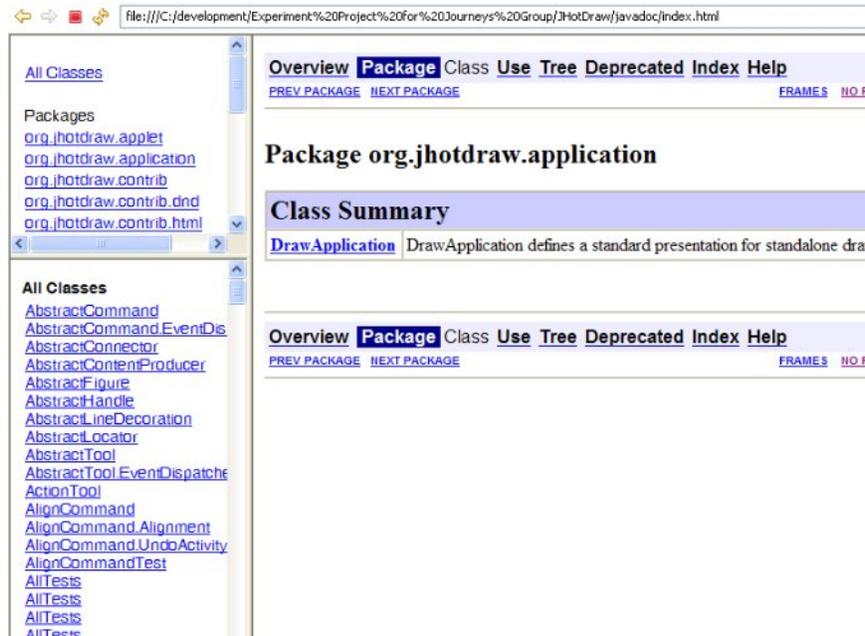


Figure 4: Example of JavaDocs Documentation

Even in the Open Source field, it appears that the majority of projects offer JavaDocs while the rest of documentation is scarce to say the least. Some of the examples include Eclipse and JHotDraw. JHotDraw contains about 900 files of JavaDocs in standard html format.

JavaDocs is aimed at static structure documentation of the components and metadata of a Java based OOP system. JavaDocs is created for systems as a whole.

2.3 Tools with higher degree of similarity

Many research efforts resulted in tool creation. Tools close to our implementation, exhibit differences and similarities. Sometimes the tools are targeting specific usage or scenarios and the difference in intent have an influence in the creation of the tool. For instance, tools intended for documentation need to ensure communication among developers. That requires features for seamless communication of locations and the sets of locations among developers. In some cases, particular implementation decisions are

dictated by the intent of the tool. For instance, a tool aimed at managing smaller code bases could afford to reparse the full code before any utilization of the tool. For larger code-bases this might result in unacceptable wait times therefore rendering the tool unusable.

2.3.1 **JTourBus** (Oezbek , Prechelt)

According to the authors, *“JTourBus is an Eclipse plug-in for navigating source code based on the idea of tours. Tours can be created using JavaDoc-tags (technically annotations work as well) supported by a little UI”*

Similarities: It is targeted as a documentation tool. It is a tagging system for important location in the code. It is easily shareable because documentation is stored with the source code thus it does not have to be managed in another documentation CMS. It also collects the tags in collections, *“one for each important aspect of the design”*.

Major differences: It belongs to the static domain. According to the authors, this is for documenting *“small or mid-sized projects”*, while we intend to target Journeys for mid to large size projects. It is static in the sense that it does not have the concept of dynamic live concerns. The plug-in loses its model between Eclipse sessions. It is meant as a tour of the source code not a live execution of the system. It is not considered to be included as part of the development process. It does not provide system state observation facility at specific locations. A tool aimed at created slide-show type presentations would put more accent on presentation and formatting than a tool intended for regular developer consumption.

2.3.2 **TagSea** (Storey, Cheng et al.)

According to the authors, *“TagSEA is a framework for tagging locations of interest within Eclipse... TagSEA allows you to: Tag locations of interest with keywords, Add date and author metadata, Filter your tags, Navigate back to places you've tagged, Create shared tags stored in the code or private tags stored in the workspace”*

Similarities: It is targeted as a comprehension tool. TagSEA uses customized Java annotations residing in the source code, and thus being shareable across the team. It is a generic tag management system that can be used for code comprehension. It is easily shareable because this documentation is stored with the source code thus it does not have to be managed in another documentation CMS.

Major differences: It belongs to the static domain. It is aimed to be used during code discovery process not as a documenting tool at the time of code creation. It is a generic tag management system that can import tags from other markers but not transform the tags into other markers (once the import is finished the tags are disconnected and changes to one collection do not reflect in the other. It is easily shareable in the sense that one feature stores documentation within the source code thus it does not have to be managed in another documentation CMS. It has no concept of tag ordering. It can import tags from other markers (TODO, Breakpoints etc) but can not transform TagSea tags into other markers (one way transformation). It does not have the concept of dynamic live Journeys.

2.3.3 Tours or Enhanced Routes for TagSEA (Cheng, Desmond et al.)

According to the authors, "Tours is an Eclipse IDE plugin that allows programmers to create and present technical talks about their work to a programming audience. Creating and presenting with the Tours system is similar to using a general presentation tool like Microsoft PowerPoint, including the notion of adding special effects. But in addition to slides, a tour (a presentation created by Tours) can include pieces of source code, which follows from the Routes concept and similar notions described in the previous section, as well as user interface elements in the IDE that can be manipulated by the presenter during the talk."

Similarities: It groups and orders locations and notes.

Major differences: It is geared towards slideshow-type presentations. It does not have the concept of dynamic live concerns.

2.3.4 Pollicino (Guzzi, Hattori et al.)

According to the authors, *“Pollicino is a plug-in that brings a simple, effective, and non-intrusive solution for bookmarking the code: adding links/pointers to locations in files (text, source code, XML, etc). It preserves the existing concept of text bookmarks while offering more user-friendly features to create, delete and manage the bookmarks“*

Similarities: Locations in the code (stones) are grouped in collections. It is geared towards code comprehension.

Major differences: It belongs to the static domain and has no dynamic concepts. It is aimed to be used during code discovery process and not as a documenting tool at the time of code creation. *“It encourages developers to bookmark artifacts while investigating the source code”*

Not considered as a development process enhancement but rather like a feature where developers might or not share their bookmarks (stones) as they reside outside the source code. In Software Journeys, the bookmarks are not inside the source code but they are treated as source code and stored in the source control system.

2.3.5 Feat (Robillard)

According to the authors, this tool is *“An Eclipse Plug-in for Locating, Describing, and Analyzing Concerns in Source Code”*

Similarities: Concerns are coupled with features (that seem to be termed as software enhancements). From their point of view, a Concern is a collection of "relevant elements". That means that the concern is similar to a Software Journey. The "relevant elements" seem to be metadata elements like methods or lines. It provides static ordering of collections.

Major differences: It belongs to the static domain. It is designed to tag metadata (methods and fields) not individual lines of code. It is intended to be used during code discovery process not as a documenting tool at the time of code creation. That is a problem in the case where methods contain hundreds of lines of code with a large noise to information ratio (unfortunately quite common case). While field tagging is interesting, it does not help during dynamic run of the application. It does not have the concept of dynamic live concerns so it belongs to the static domain.

2.3.6 **ConcernMapper** (Robillard, Weigand-Warr)

A simplified tool evolved from work on FEAT.

According to the authors, *“ConcernMapper is a plug-in for Eclipse that allows you to organize fields and methods into arbitrary modules called concerns. In brief, ConcernMapper allows you to reorganize the modularity of a software system in a way that suits your needs, without altering its "official" structure or behavior. It also allows you to keep a permanent record of the code associated with various concerns.”*

Major differences: It belongs to the static domain.. It is designed to tag metadata (methods and fields) not individual lines of code. It does not have the concept of dynamic live concerns. It does not provide ordering of collections. It is aimed to be used during code discovery process not as a documenting tool at the time of code creation.

It is important to notice that all these mentioned tools belong in the static area.

2.3.7 **Detailed comparison of Software Journeys with other tool based solutions**

The detailed comparison information is shown in Table 1: Detailed comparisons for first set of characteristics and Table 2: Detailed comparisons for second set of characteristics.

Approach	Functions, stated purpose	Dyna-mic	Ordered locations	Direct source editor marker change	Line-block of code level edit	Exports/Imports markers	Persists on work-space exit	Marker storage
Journeys	Targets productivity increase when used as part of the dev. process	Yes	Yes - implicit	Yes	Yes	Yes - Needed only for exchange with other systems	Yes	external of source code. Internal in project
JTourBus	<i>"plug-in for navigating source code based on the idea of tours"</i>	No	Yes-hard coded	Yes	No	No (not needed)	No	internal (with the source)
TagSea	<i>"Framework for tagging locations of interest within Eclipse"</i>	No	only for routes (unavailable feature)	Yes	Yes	Yes (only for Routes). not available	Yes for tags and waypoints/ unknown for routes	external for routes
TagSea Tours	"Organize files and waypoints, and create transitions/effects to present work in classrooms/conferences"	No	Yes	N/A	Line level	N/A	Yes	N/A
Concern Mapper	<i>"extremely lightweight and easy-to-use cousin of FEAT"</i>	No	No	Yes	No (in Project Explorer)	Yes. Non XML extension	Yes	external
FEAT	"To give developers a single, focused view of the subset of a program implementing concerns."	No	No	No	No (in Project Explorer)	Yes	Yes	external

Table 1: Detailed comparisons for first set of characteristics

Approach	Hints for meta data info	Multiple landmarks can reside at the same location (different Journey)	Sharing	Comments	Experiment	Synch on refactoring	Multi level	Productivity improvement, quantitative results
Journeys	Yes	Yes	via source control			Yes	No	Yes
JTourBus	No	No	via source control		JHotDraw	Manual /automatic depending on usage	No	Yes
TagSea	No	No to for tags. Unavailable for routes	via manual export/import	Routes was not present during experiment. <i>"Routes are no longer supported. The initial experiments showed them to be too volatile and unreliable in their current form"</i> .		Manual (check all files in workspace)	Yes. Using . as level delimiter in strings	No
TagSea Tours	N/A	N/A	N/A		N/A	same as TagSea	N/A	N/A
Concern Mapper	No	No	via manual export/import	no extra metadata can be added to the elements in the concern	JEdit etc	Yes	No	No
FEAT	No	No	via manual export/import	no extra metadata can be added to the elements in the concern	JHotDraw JEdit, etc	Manual	Yes	Yes (qualitative) Tool/cont. group implement the change

Table 2: Detailed comparisons for second set of characteristics

Chapter 3

Perspective and Novelty

- This work is considered from the perspective of a technical development manager. This perspective is an encompassing view, considering multiple aspects of software development in correlation. One of the important parts of the correlation consists of teamwork and processes including capturing, storage and communication of information.

- All surveyed previous work, looked at possible solutions in isolation, and did not explore where and when the solution fit in the overall development. This gap was also pointed by a recent study that concludes *"Research should focus on providing an ecosystem that supports the entire tool chain"* (Parnin, Orso 2011). The same study, (Parnin, Orso 2011) notes that *"In general, without a clear understanding of how developers would use these techniques in practice, the potential effectiveness of such techniques remains unknown."*

- This approach offers a tooling solution for known software development practices
- This approach offer a solution to documented developer demands (captured during previous studies)
- This approach designs a new form of documentation
- This approach allows a developer to consume this documentation in a live manner. This was never done before
- This approach uses a dynamic form of documentation topic ordering
- This work perform experiments based on this novel approach
- This approach allows for custom merging of Journeys and selective disabling of Landmarks
- The Journeys are based on work items in the project plan which is an existing and reliable artifact in the SDLC

Chapter 4

Solution

4.1 Usage Scenarios

Fred works as an intermediate level software developer working for a large established IT company. He is part of a team of twenty software developers. The team also includes analysts, specialized QA resources, support staff an architect, a project manager and a technical development manager.

The team develops supply chain software that is used by large manufacturers and typically manages several hundred millions of part versions.

The software development process is based on Rational Unified Process and it is reasonably well documented and followed. The process includes the use of Eclipse-a standard IDE (Integrated Development Environment), source and defect control software, use cases, test cases, coding standards, unit testing and code reviews. Normally the team produces a major release every year, a half-year enhancements release and fix pack releases on demand.

The current project plan contains a number of line items for the current release. One of the line items requested by one of the manufacturers is a new feature called “Manage Training Information Feature”, where the supplier could add training materials for the supplied parts. A use case and a test case called ‘add install tutorial’ have already been produced as the software is in the inception phase of the SDLC (Software Development Lifecycle).

As a result of a preliminary sizing exercise it is estimated that this new feature should take roughly two weeks. That includes coding, documenting, unit testing and rework resulting from unit testing. Out of this coding takes about 90% of the allocated time, so about nine pd (person days) of effort.

The new feature allows the supplier to associate different classes of training information like multimedia with a product. During the management of this association (adding new information or newer versions of updated information, editing, deleting, moving) a certain number of business rules and validations (depending on the type of information) are also executed. On information update, the supplier is prompted for backward compatibility information.

During coding, documentation and unit testing phase Fred creates a new Journey called “Manage Training Information” consisting of about ten landmarks.

The landmarks are code locations that Fred considers the most relevant in comprehension of the system for this feature. They are augmented with documentation mainly describing the state of the system when program control reaches those locations.

One of the landmarks is at the beginning of data validation, several corresponding to non-trivial business rules, a couple of landmarks before saving the data in the cache and a couple of landmarks after retrieval from the persistence layer and before committing to the persistence layer. A couple of other landmarks are placed in other locations of special interest.

One of the goals is to describe non-obvious system state and behavior code sections with emphasis to feature centric documentation versus static structural documentation. Fred follows the guidelines and for each landmark he enters short paragraphs to maximum one page of documentation and two lines to describe two selected class fields. He adds about one to two landmarks per development pd. The landmark number and abundance of documentation is correlated with the complexity of the feature and code.

During the code review phase, Fred uses the landmarks to bring the review team members up to speed and, as a result of review board member questions he decides to add one more landmark.

The Journey and all the landmarks are automatically saved on workspace exit in an xml file that is under a source control directory. When he checks his code in, a non-conflicting merge is performed on the xml file because his Journey is disjoint from other features/Journeys other developers work on.

Two years down the road, several clients ask for a major enhancement to the training information process with extended support for structured information.

This new type of support information like AutoCAD, Revit and other proprietary formats contains references to other parts produced by the same or different suppliers.

The clients are interested in using those references to establish relationships with other training information and other parts in their system so overlapping training is not performed and to ensure no obsolete training material is used to train the assembly line workers. The feature is deemed important, coined “Cross Training Information Feature” and scheduled for the fourth major software version of the product.

As the associated information is structured, it contains references to other parts by the same or different suppliers. The software needs to maintain an extra layer of relationships between parts based on the training information affinity. The supplier is informed when a certain part training info contains training related to another part they provide and asked to update the other part training if necessary. This new software feature will allow the manufacturer to run consistency checks and training program creator wizards that collect and intelligently merge together all the training for all the items used in the manufacturing of a certain product.

Fred was moved to another team in the same company, and is fully utilized for other development tasks, so Laura, another intermediate experienced developer, is assigned the new project line item representing the “Cross Training Information feature”.

Laura, who has never worked in this area of the code, is not familiar with the implementation. She checks and finds out that her new feature is related to the feature developed by Fred.

She loads the code in her development environment, brings up the Journeys view, selects the “Manage Training Information Feature” Journey and clicks run journey. The system starts running and Laura follows the “Manage Training Information Feature” test case.

The system stops at the first landmark in the Journey and Laura reads all the associated documentation provided by Fred. She also uses the view state facilities provided by Eclipse to look at some of local and class variables, calling methods stack etc to get a more comprehensive picture of the implementation. She does not have to limit herself to just the variables documented by Fred but she can explore other dimensions she would find useful. She continues to the second landmark and so on until the end of the Journey. This information augmented by other system documentation allows Laura to get a decent understanding of the implementation Fred provided.

She uses the information to decide the best locations where she should implement her new code, how much code she can reuse, parts that need modifications etc.

Laura also creates her own new Journey “Cross Training Information Feature” and Landmarks following the same guidelines as Fred. As she is reusing part of Fred’s code she adds a couple of landmarks to the existing code including one landmark at the same location as one of the landmarks added by Fred. The information entered by Laura is also shared, reviewed and discussed as part of the code review process.

This information will serve for future development.

4.2 Problem description

There are very few techniques available to narrow down the exact line(s) of code where change is needed.

The best-case scenario relies of the developer’s “familiarity with the code”. In other words, the developer is already familiar where that file and line of code is because it is the same developer that coded the feature.

When the developer is not familiar to the code, surrogate techniques are trying to narrow it down by reading the requirements documents to extract the behavior information, followed by a search for the corresponding high-level design document followed by the corresponding detailed level design document(s). Once those documents are identified, the next level is to try to extract out of these documents sections that provide clues of where the sought code lines are. The best clues usually come from class diagrams or component diagrams and are generally weak clues for several reasons: This type of documentation is rarely up-to-date, it does not have the required granulation, is disconnected from the code (Latoza, Venolia et al. 2006) and is not aimed as a documentation to line-of-code mapping.

In conclusion, the software development process fails the developers in one of their primary tasks.

This problem grows with the size of the system since the bigger is the haystack the harder is to find the needle.

In many cases, the resource expenditure for finding this mapping might surpass the effort to code the changes.

The main issue in here is that the original developers that has the knowledge is either unavailable to continue enhancements in that code area or if lost that information due to the passes of time.

4.3 Scope

This approach defines a theoretical core scope and a theoretical extended scope. The theoretical core scope is the scope where claims are made.

In addition, the tooling has its own scoping domain that would be considered separately

The extended scope covers hypotheses and some aspects of the experiment where the approach might offer value however, no claims are made.

4.3.1 Core scope

- Mid to large codebase software
- Experienced development resources having multiple years of non-volunteer development experience
- For profit software development (this includes non-financial profit)
- Teams that follow a development process

4.3.1.1 Extended Scope

- It is probable the same approach might yield value for small code base software however no claim is to be made in this case
- It is reasonable to believe the same approach would yield value for junior resources however, the focus and the experiment would not cover this area. We believe an efficient team would carry a small percentage (around 10%) of junior resources
- It is possible that a team that does not follow a development process will benefit from this approach. We don't include this in the core scope because we believe that a team that does not follow a project plan, does not capture requirements and does not use test cases should firstly invest in this infrastructure for a higher return on their investment
- It is possible that a team that does software development as a learning activity might benefit from this approach. Since the incentives are different from a standard industry-production environment the team might not have the discipline of consistently capturing this new form of documentation. That would result in low levels of commitment and trust and consequently in higher level of abandonment
- It is possible to additionally capture other type of information when the live journeys reach the landmarks. That is not the focus of this approach and would be explored only as a possibility.

4.3.1.2 Tooling Scope

This approach is generic in nature. The demonstration of this approach will be done by designing and building specific tooling. This is common practice for research work and does not create any loss of generality.

We have chosen to implement tooling for systems developed in Java programming language

We have chosen to implement the tool as an Eclipse plug-in

The reasons for this selection are that java is platform neutral and Eclipse is considered an established, popular, and extensible platform.

There are provisions in the development environment and in our approach to extend this solution to other programming languages and other types of resources that can be tagged.

4.3.2 Requirements

4.3.2.1 High level requirements

These are generic requirements that are independent of the platform, development environment, and programming language

- The solution needs to allow the creation of location dependent, context specific and feature (/use-case/test case) related documentation
- The solution needs to store the documentation in a persistent fashion as to be retained between environment restarts
- The solution should allow developers to capture this documentation during software coding phases
- The solution should allow for team sharing of the documentation without requiring additional operations (like import, export)
- The solution should allow developers to embark on a live journey and allow consumption of the captured information when the journeys reach the landmarks
- The solution should not require any additional infrastructure
- The solution should allow the communication of metadata to other development environments
- The solution should be able to integrate in a development process

4.3.2.2 Detailed requirements

These are specific requirements that might be dependent of the platform, development environment, or programming language

- to be able to aggregate a sequence or a number on landmarks in a collection (Journey) based on a feature identifier, use case or test case

- to be able to select a specific journeys and retrieve the sequence of landmarks associated with that journey
- to be able to ‘map’ a journey so that a live execution of the journey would stop and communicate landmark specific information
- to be able to embark on a journey in a live manner and present the captured information when the execution of the journeys reach the landmarks
- to be able to store journeys and landmarks in a persistent fashion
- to be able to share journeys and landmarks using source control systems
- to be able to associate field names and state with a landmark
- to be able to have different landmarks for the same location for different journeys
- to allow capture and display historical run information
- if needed for performance: to be able to cache and provide a synch with the landmarks facility
- to support scenario variants for the same journey
- support automatic child-parent migration of documentation when necessary

Optional requirements

- to be able to link other artifacts with the journey
- to be able to start a journey using a pre-recorded scenario and stop at the pre-established landmarks

4.4 Solution

Instead of trying to apply industrial espionage techniques to determine information about one’s own code it is better not to discard that information to begin with. Of course, the (Chikofsky, Cross 1990) taxonomy paper on software reverse engineering takes care to point out that the connotation of espionage in hardware should rather be understood as usually forgetfulness recovery in software.

Not unlike previous research solutions, we will create and approach and tooling to create and later to consume a new type of feature related documentation pertinent to selective code locations

We also advance the hypothesis that this particular method and tooling would permit an increase in productivity.

In order to verify this solution, we design and execute an experiment to gauge the effect on efficiency.

- Tool Development Methodology

We will approach this endeavor following a systematic workflow based on an established process. The phases will include analysis, design/architecture, feasibility and non-functional requirements, and proof of concept implementation.

As developers are subjective by definition and their approach would have a level of variability, some of the decisions will have to be made based on empirical or incomplete information. As a result, the level of confidence in certain features and decisions will not be absolute. This is normal and it provides a basis for further discussion and improvement.

- Analysis Methodology

We consider the specific requirements, previous approaches, and shortcomings. We identify specific goals for a successful system and identify possible alternatives and specific challenges. In the end, we select one of the options for design and implementation. A new introduced tool usually requires increased attention. This extra effort might skew experimental results especially for more complicated features. In order to minimize this effect, we will follow the KISS principle (Johnson)

- Design Methodology

We identify the factors that could influence design decisions. We analyze options for the software persistence model and for the user interface. We follow design principles and additional considerations to select one of the existing options.

4.5 Analysis

4.5.1 Strategies for Associating Journeys with existing SDLC concepts

Software Journeys are documented during the coding phase. A developer that codes a project plan feature functions as a resource in a SDLC process.

A normal SDLC is based on tasks from project plans. There is quite an array of terminology to name these depending on the framework used for management. For instance, PRINCE 2 uses term work-item. In some cases, this is called unit of work. In other cases for large mature organizations like IBM, a line in the project plan is called a line item. Sometimes even the same organization employs a multitude of terms like feature, enhancement, change request etc.

For clarity purpose, we will use the term feature to represent a unit of work from the project plan.

Most of the features correspond to one or many use cases. Some organizations use the term scenario instead of use-case, which is usually associated with RUP.

For clarity purpose, we will use the term use-case to represent both, a use case or a scenario..

Usually features that are small and take less than two PW are assigned to a developer and usually are associated with one use-case. Larger features are usually split. For iterative development processes, it is easier to manage resources when features do not extend beyond one development iteration duration.

Each use case is characterized by a primary (or basic) flow and sometimes alternate and exception flows.

Once the use cases are coded, a unit-testing phase takes place. Most of the case that is informal however is some cases developers are required to capture the unit-tests as part of the development process. In some cases, these test cases are written using an automatic testing framework like JUnit. Automated unit-test-cases, can be grouped in test-suites and run as part of automatic regression testing.

Subsequently, the product is transferred to quality assurance group that creates and executes test cases for the features. These test cases can be described in a text form and depending on the team and the feature, some form of automation might be used. Test cases case can also be referred as test scenarios.

Regardless of terminology, our approach associates a journey with a feature executed by a developer.

Since the software development uses various standards, we need to be flexible and should not mandate specifics for this association. In order to accommodate this diversity we provide association guidelines as to associate Journey either with features or with use-case or with use-case flows.

The feature-based association is quite natural as features (and their identifiers) are also used for time reporting thus are managed very closely. A simple implementation could use the same name for both the Journey and the feature name.

Should the use-cases be clearly documented and a higher need for granularity be required the best option is to associate Journeys directly with use cases.

In many cases, the feature to use-case is a one-to-one relationship so this separation would be a moot point.

Depending on the level of detail in the process, the Journeys can be created for selective flows in the use case. This would offer the maximum granularity and be useful for organizations where the use-case granularity is very coarse.

Some development processes follow a test driven methodology in which test cases are developed before coding. In these cases, the best level of association is directly with the test cases. That applies to all types of test cases. The test cases have the advantage that they are usually very clear and specific.

It is also possible to post-associate captured journey with test cases developed subsequently. Many test cases take the names of features so the link would be quite obvious.

Regardless of the granularity, the development manager would have to choose an association option, and document that in the development process. Furthermore, this information should be disseminated

clearly within the development team so it creates a good level of compliance for documenting developers and a high level of trust for the developers consuming this form of documentation.

4.5.2 Grouping

One important component in completing software features is the need for grouping locations. According to (Ko, Myers et al. 2006) “*the environment must also provide a reliable way to collect the information the developer deems relevant*”. Obviously, the collection has to pertain to the work item being coded.

Marking code locations is already feasible using normal features exposed by established IDEs. Eclipse lets developers use several marking tools, however grouping is not supported. Most of the research in this space offer that capability. Some other tools like (Guzzi, Hattori et al.), TagSea (Storey, Cheng et al.), Feat (Robillard) and ConcernMapper (Robillard, Weigand-Warr), etc offer flavors of grouping.

Simple grouping or multilevel groupings are potential options to choose.

New tools require a learning cycle and the more complicated the tool is, the steeper and longer the curve is. Our philosophy is not to choose increased complexity features without having substantial evidence that the complex feature is superior. Without that certitude, we will incline towards simpler to understand and use tools.

In order to choose one-dimension collections or multiple dimension collections, we used information derived from two other previous research tools, Feat and ConcernMapper. Both tools were developed by the same researchers, and Feat, the tool that was initially developed offered multilevel grouping. According to the authors the ConcernMapper plug-in evolved from first author’s work on FEAT.

The fact that authors did not port this feature in the next tool suggests that the non-hierarchical approach was considered more advantageous.

It is natural to approach adding advanced tooling features in incremental fashion from simple to complex, so based on this principle and results from previous work, we decided to use one-dimensional

collections. The actual multilevel grouping could always be achieved by means of namespacing the Journey names without requiring a more complicated approach.

4.5.3 Selection Granularity

Based on research done in (Latoza, Venolia et al. 2006) and (Ko, Myers et al. 2006) we decided to allow line-level landmarks as the most common relevant granulation for developers.

Some systems create large methods for different reasons. In some cases is because of performance, sometimes it comes from legacy systems and many times is just poor coding. That is very common in real-life situations. As an example in the Eclipse plug-in the view is supported by the Tree class. The method 'CDDS_ITEMPOSTPAINT' contains 549 lines of code, which in the Java perspective that accommodates 35 lines of code results in more than 15 screens at a common 6018x1050 screen resolution. In my department, one of the live systems has a method that contains more than 100 screens of code.

4.5.4 Training

As the Journeys documentation is collected during work item development of specific features, journeys will be associated with features or their components. That is an important distinction since all other tools but one, are not targeting documentation and are geared towards investigation of previously produced code.

It is important to capture the principle for the grouping criteria in the tool user manual and tool training. The developers creating code will have to respect this criterion. Subsequently, developers consuming the documentation must be informed about how landmark are grouped together so they know how to utilize them. The (Parnin, Orso 2011) study concludes: *"Developers were quick to disregard the tool if they felt they could not understand how such results were computed"*

Adhering to these principles will build trust and increase conformance to the process.

4.5.5 Information Volume

As we propose to introduce this method in the development process it's important to determine a guideline as to the number of comments to be added per feature. That is important because we need to reach a balance between documenting too little and spending too much effort on this activity. We can start with experimental results from (Ko, Myers et al. 2006) study that states that an average of thirty lines of code per task were considered relevant by the developers. By observing, the example given it clear that we have about eight contiguous sections, therefore we could use about eight Landmarks. We assess that having a landmark placed at the beginning of the journey in the program's start-up sequence, will help developers orient better in the beginning. In conclusion, we would recommend an average of about ten Landmarks per feature adjustable depending on the size and complexity of the feature.

4.5.6 Documentation Location

An important principle of this approach is documenting important code locations. One of the characteristics discussed by the most related work is the location of the documentation as internal or external relative to the source code.

Our approach has similarities with JavaDocs as both are code-documentation and consumption tools. JavaDoc is associated with code locations similar to our approach and it can be used to produce code external documentation. JTourBus, the only other documentation focused approach found during the literature review, adds the documentation directly in the source code.

One of the reasons we decided to externalize the documentation was to avoid bloating of code files. Remember that this documentation is feature specific and might be acting as 'spam' for a developer using a common section of code for other features.

The other reason for this decision comes from the (Latoza, Venolia et al. 2006) study that found that: *“There are plenty of reasons that a developer would choose to not record the information. The overhead of checking the code out, editing it, and checking it back in (possibly triggering check-in review processes, merge conflicts, test suite runs, etc.) is enough to dissuade the developer from recording the information as a comment in the code. “*

4.5.7 Documentation and life-time of software

The perennial scope of documentation should be kept in mind. We must make sure the documentation is still consumable in the future as this documentation is acquired for future use, and established software can exist for many years.

While we firmly believe in the advantage of consuming documentation while using the code in a live manner, we also see value in providing a feature for externalizing the captured information for other uses

Reliance on IDE is reasonable however most IDEs do not assure firm contracts in terms of data formats. For instance, Eclipse's marker formats, might change in future versions so IDEs might not be able to consume the documentation. Thus, we have a need to make available the documentation in a standard format that we believe can withstand time. Based on this, we decided to add a facility to export the documentation in an XML format and to import it from that format. The Journeys and Landmarks will be automatically persisted so no use of import/export features would be required in that respect. The import export facilities would only be necessary when changing the version of IDE or transferring the information to non-Eclipse environments.

4.5.8 Non-runnable code resources

While normally, we regard source code as runnable, that is not always necessary the case. Some high level languages accept declarative forms that do not produce microprocessor instructions.

Most notable cases are interfaces and abstract methods.

Since they do not contribute executable code, it seems unlikely that those would be good candidate for Landmarks. Assuming we would set them as Landmarks, they would never be used and their associated documentation would not be visible when the program execution reaches those positions.

Normally, the prescribed solution would require developers to create this documentation for each feature they implement. The only way to use an interface or an abstract method would be to implement it in a derived class. Since the actual behavior is contained in the actual implementation, it would be normal to set the Landmarks in the concrete method.

As an exercise that could yield value, we decided to allow Landmarks on Abstract classes and automatically create derived Landmarks for the same journey. The developer creating the Journey might decide to keep either the base Landmarks or just the newly created ones.

4.5.9 Extending OOP concepts to documentation

Normally, documentation is present in a simple string-like form. At times, it can also carry format information. In our case, this documentation is coupled with code locations, similar to a GIS system where information is associated with a geographical coordinate.

Since the location is an OPP code location, this connection to an OOP artifact, opens the possibility for further exploration. The main questions associated with this concept extension are:

- Can some of the OOP concepts be extended to this type of documentation?
- Is the extension technically feasible?
- Can we envision cases where this extension might be useful?

For the purpose of this exercise, let us assume that the documentation is created at the method level.

Arguably, polymorphism is the claim to fame of OOP. Inheritance and method overriding are needed for enabling polymorphism.

Inheritance establishes a parent to child relationship for classes and implicitly for contained methods.

In our approach, both the parent and the child method can carry documentation. In that case, by extension, let us call the parent-documentation to the child-documentation a parent-child relationship. Obviously, that would be a directed relationship.

OOP inheritance is a static relationship that has dynamic implications because the program decides what method to execute at running time. That is called dynamic binding.

In cases where a child does not implement a specific parent method, the child ends up by executing the code contained in the parent method. This happens as if the code of the parent method was migrated to the child during the actual execution.

Let us look at the same scenario when the parent class method has Landmark documentation and the child class method does not supply its own documentation. Based on the OOP behavior we can envision a case where the documentation is "dynamically bound" and the parent documentation migrates to the documentation deficient child.

So far, we provided a 'yes' answer to the first question.

It is possible to determine all the implementers of an interface and all the overriding methods of specific Landmarked method. In consequence, this extension is technically possible. Considering that one interface might have hundreds of implementers, this might not be performing, nor desirable. A good middle ground solution might be to implement this new documentation behavior only for interfaces and abstract methods and to stop the chain of migration when we reach another already landmark in the same Journey. See Figure 5: Late binding of landmark documentation achieved by migrating parent method documentation to child method node for an example of migration of the documentation. In this figure, the original Landmark documentation is depicted in the top left note, and the migrated documentation in the red-font notes below.

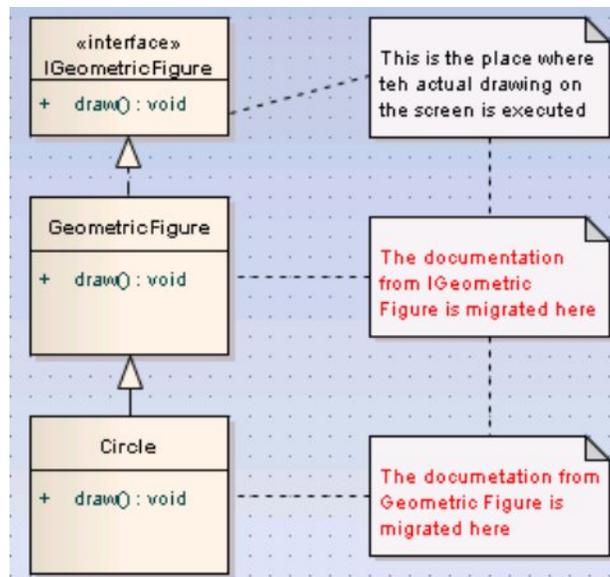


Figure 5: Late binding of landmark documentation achieved by migrating parent method documentation to child method node

- Can we envision cases where this extension might be useful?

This extension of OOP concepts would add additional generated Landmarks and generic documentation in some specific cases and it should be regarded as an additional tool in the panoply. Studies like (Widowski 1986) discuss novice and expert techniques. They found that experienced developers have more techniques at their disposal. In addition to that, they are more adaptable and flexible in using one technique or tool based on the task. Experienced developers demonstrated an opportunistic behavior when faced with the challenge of software comprehension. It would be up to the original developer to decide to keep both Landmarks or just one of them.

For the scope of this experiment, we decided to manually simulate this creation during documentation time and used these ‘late bound’ documentation during the consumption of this documentation.

It is interesting to note that this feature is not present or supported by current documentation systems like JavaDocs. Such a feature might not be beneficial to JavaDocs mainly due to its static consumption.

4.5.10 Optional features

4.5.10.1 Heap Snapshots

We could extend this solution to capture heap snapshots. Consuming the snapshot information could be useful because, in principle the developer could compare previous run local variable values to his own execution values. Obviously, the more dynamic and complex the system is, the more likely is that the two sets of execution-runs values could be very different.

Overall, that might or might not provide value.

This feature would require not only capturing of the value sets but also the infrastructure needed to store, modify, and manage these value sets.

A previous research (Ghabi, Egyed 2011) states that “*Unfortunately, capturing and maintaining traces is a largely manual activity*”

Based on personal experience I believe convincing value would not be produced for large segments of software development. During my career, I worked on hundreds of projects of diverse software development for diverse companies including many industry leaders. For all but one of these projects, trying to control the system input in order to generate perfect reproducible stack traces was considered impractical. Not all these projects were using exclusively user input but also additional input sources. Most of these projects, dynamically read and wrote from a multitude of live databases that in many cases were themselves under development.

Freezing these databases was considered impractical because the time and effort of capturing and management of snapshots.

In (Ghabi, Egyed 2011) Ghabi, A. and Egyed state *“Whether requirements-to-code traces are captured manually or through the help of such heuristics, the quality of the traces is unpredictable. Even if traces are captured by the original developers and are thus of high quality, these traces may deteriorate over time (with code and requirements changes) unless they are explicitly maintained.”*

The time needed to capture, synchronize and manage these inputs was many times longer than the time interval between two successive builds. For instance, currently, at Environment Canada, most of the products consist of Maps that render differently depending on date, content of several internal and external databases and the rendering capabilities of the rendering workstations so heap captures would differ for each run.

The only initially “perfectly observable” system I ever developed was a large telecom system called “Stentor Settlements”. In this system, all the paid long distance calls crossing Canadian territory, were divided between different telecommunication companies that carried the calls over segments of their physical network. The actual financial amounts were consolidated monthly for each phone line owner. The system was a large mathematical formula ran on millions of records. During testing, it was discovered that the total calculation time was longer than the billing cycle therefore the system was unusable. An executive decision was made to randomize the distribution of the money for calls under a certain amount. In the end, this system became “random” and heap or execution trace captures would not have been consistent in time.

Storing internal values from past executions would be interesting as an extension of standard test cases. Furthermore, automation could be provides in order to test the system not only for the output state but also intermediate state. That additional feature might provide for a faster defect resolution. This extension belongs to the realm of software testing and could be considered as part of future explorations.

4.5.10.2 Pre-recorded scenarios

As we mentioned during the discussion in associating Journeys with SDLC concepts it is possible to associate Journeys with defined test cases directly or indirectly by following the feature to test-case association. This would offer the advantage of knowing exactly what sequence of steps should be followed after the start of any journey. This behavior is not implemented in the tool however it could be easily be supported by recording the test case(s) name in the Landmark documentation.

Normally a feature has one of several primary use case and several negative test cases which usually test the boundary conditions.

4.5.10.3 Hyperlinking non-runnable resources

Multiple artifacts are using during the normal process of software development. Some of them are part of deployed system and out of them, some are necessary for the running of the system.

Out of that, beside the normal category of runnable code, some artifacts are non-runnable. Some classic examples are resource files, xml descriptors etc. They are modified as part of code changes and it might be useful to add them to the Journey collection. Some of the research solutions like (Guzzi, Hattori et al.)allow tagging of non-code resources.

There are two distinctive options. The first one is to implement tagging of such artifacts and since they are non-runnable, ignore them during the live execution. The alternative is not to tag non-runnable artifacts.

Our approach is based on the principle that filtering of code locations based on feature and on execution stage, simplifies most complex code-bases. Consistent with the same principle we believe it is better to associate these non-runnable artifacts with the journey, as part of the locations where such resources are used. For instance, when we use the JHotDraw graphic editor application (Gamma,

Eggenschwiler et al.) one or the Journeys might use some image files. One of the places on that journey would load the image from a .gif resource. If this is deemed important, we can create a Landmark and document it with: “the circle image is read from the imageName.gif” as in Figure 6: Landmark properties.

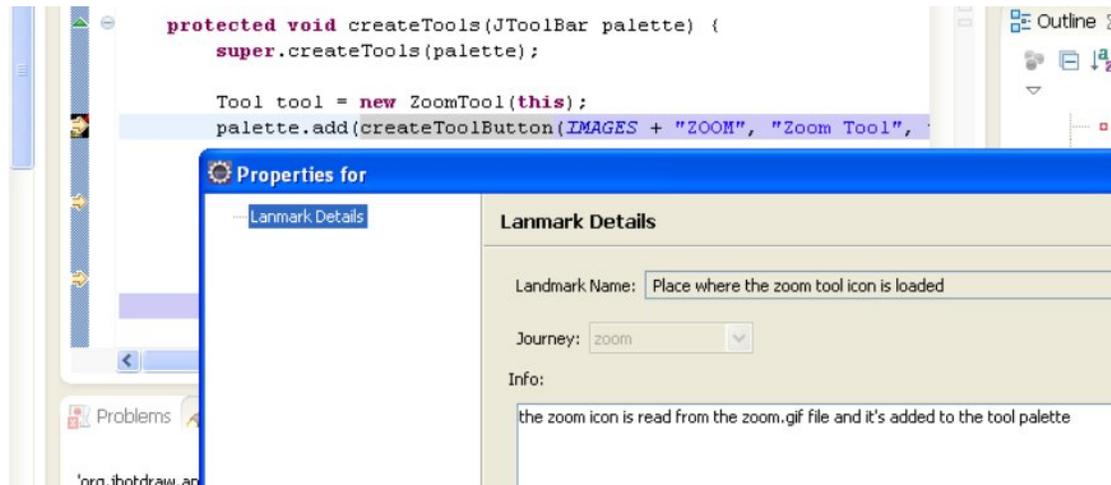


Figure 6: Landmark properties

This way developers are informed of the utilization of the imageName.gif file only then this resource is really used.

That being said, some developers might prefer having the resource directly part of the Journey collection and the tool could allow future extensions to accommodate that should it be requested by developers.

4.5.11 Alternate Journeys

Let us consider the following two scenarios:

An ‘Iron Chef’ goes on an international Journey called ‘The Art of Healthy Cooking’. One landmark on this Journey is Paris.

An Art professor goes on a worldwide journey called ‘Medieval Art and Architecture’. One landmark on his Journey is also Paris.

In software world, these are different use-cases.

Despite the same location, being common to both journeys the agendas will be disjointed. We might as well call the first landmark Gastronomic Paris and the second landmark Medieval Paris because the experiences will be different.

For the same reasons our approach will allow the creation of multiple Landmarks for the same location and we will not reuse landmarks among Journeys.

On the other hand, the art professor might be flexible and make changes to his journey based on external factors like weather for instance. In case of bad weather, he would go to Rome and take pictures of the edifices for his upcoming book.

If we consider the Paris sojourn the basic use case, the Rome landmark can be considered as an alternative flow for the ‘Medieval Art and Architecture’ journey.

We could create these as two different Journeys called “Medieval Art and Architecture including Paris” and “Medieval Art and Architecture including Rome” however we would have to re-document all the rest of landmarks.

Creating different variations to one journey would be preferred in this particular scenario.

A possibility would be to design a User Interface to mark alternate Journeys as children of the basic journey.

Another way this can be achieved by the chosen implementation is to add both Paris and Rome landmark. Depending on the execution scenario, the live Journey will include just Paris, Rome, or both landmarks. It would also be possible to highlight somehow landmarks that are used only on alternate use cases.

4.5.12 Synchronization

Normally resource synchronization is to be considered during the design and implementation phases. Most development processes suggest considering high-risk items during the analysis phase. Synchronization presents special challenges and can affect the overall solution feasibility.

The more resources have to be changed in unison the harder is to control synchronization. This task becomes very time consuming when the volume of resources is very large. That is true regardless of the development environment.

In our case, we need to build software able to contend with large amounts of code, anywhere for several tens of thousands to millions of lines of code

It is reasonable to parse in real time several thousands of landmarks and journey records, identify the source files, and update the interface.

In principle, it is possible to parse the code-base looking for landmark annotations and update the Journey-landmark views. While that is possible, doing this for large code-bases will be unacceptably slow. It would probably take many minutes and certainly would not be possible in real-time. Changing one source file at a time should not be an issue because we can parse that particular file. The issue would occur when starting the software or loading another workspace. Clearly practical considerations require us to maintain a model outside the source code.

To probably design this based on classic model-view –controller patterns we will have to keep a model with all the journeys and landmarks in memory and decorators for opened source files.

Some of the other solutions that chose to implement in-code tagging or other form of annotations will need to resort to manual synchronization mechanisms.

The Eclipse development environment introduces additional elements and some result in new challenges.

One of the features offered by Eclipse is the marker facility. This facility offers libraries to create developer-defined tags that can be associated with code locations. An advantage of these tags is the automatic handling of persistence and some of the synchronization. These markers are the base for core elements of the IDE like bookmarks, todos, problems, breakpoints, tasks etc. They are appealing to use however, they are static, and their attributes cannot be defined dynamically.

In our case, landmarks carry context sensitive class variable attributes and they are not known in advance.

The main challenge posed by the environment stems from its design as most operations are carried out in separate threads as asynchronous operations. Refactoring operations are notoriously unreliable in Eclipse and the asynchronous tasks mechanisms have been redesigned multiple times during the last decade. While these operations execute, they might leave temporary orphans. In some cases, depending on the relative speed of threads, these orphans are permanent.

Most of the marker-based concepts exhibit that behavior and, in some cases, the interface is not updated. The validation and synchronization is done usually upon usage by calling the `isValid` method on these objects. Should the object be orphaned the marker is removed and it is up to the developer to inform the user. This behavior is by design.

The Software Journeys implementation separates the model from views and uses automatic synchronization.

4.5.13 Extensibility

There is a potential usage of this documentation by other tools for analysis, formatting etc. Let us say that a graphic program is created by a team of developers and different developers create functionality for different shapes, thus creating a rectangle manipulation Journey and a circle manipulation Journey. A tool could read the xml file containing the journey documentation and present a side-by-side comparison of Journeys. That could be used to assess consistency of implementations.

4.5.14 Navigation

Navigating between code locations is significant as it is significant especially while exploring non-familiar code. According to (Ko, Myers et al. 2006), “*The total time developers spent navigating was 35 percent of the time spent developing code*”. Some of the solutions like Pollicino (Guzzi, Hattori et al.), ConcernMapper (Robillard, Weigand-Warr), base TagSea (Storey, Cheng et al.), Feat (Robillard), etc] that grouped collections of code locations have not indicated any navigational aids like a navigation order. Others, like JTourBus included hard-coded ordering. Our approach not only should offer implicit ordering

but should also automatically navigate to correct locations during the live execution of the software system.

As an additional feature, we provide a method to display live execution order and live Landmark visitation information.

Besides the normal flow of execution, this live display can provide information about landmarks that have not been visited.

That would accommodate both alternate-flow use case scenarios and exception-flow use case scenarios.

In addition, unvisited landmarks can provide clues to developers working on defect fixing tasks.

4.6 Design

4.7 Design principles

Design to optimize the use of existing resources

Design for usability

Design for clarity and minimal additions to existing user-interface

Design for flexibility and extensibility

Use an incremental approach to minimize user confusion

Use predictable behavior

Design for robustness

Design for adoption

4.8 Design Decisions

4.8.1 Creation of Landmarks

According to (Storey, Aranda et al. 2011) a possible solution “*is to propose lightweight, low-risk techniques*”. Based on this observation we decided to use a lightweight approach that maintains the full flexibility of the current environment and allows developers to continue using their favorite methods without imposing rigid structures. One of them is to propose lightweight, low-risk techniques

We decided to create two iterations of the design and implementation and collect feedback from the first iteration to be used in the second iteration. Only two participants were available for feedback.

The initial iteration included some options designed to elicit feedback.

The first iteration and allowed for creations of landmarks from the main menu, tool bar, source file view, decorator bar, source file and Journeys view. See Figure 7: Creation of Landmarks from the main menu and decorator bar and Figure 8: Creation of Landmarks from the main toolbar and Journeys view. Both the decorator bar and the source file submenu were accessible by right-clicking on the decorator bar and source editor.

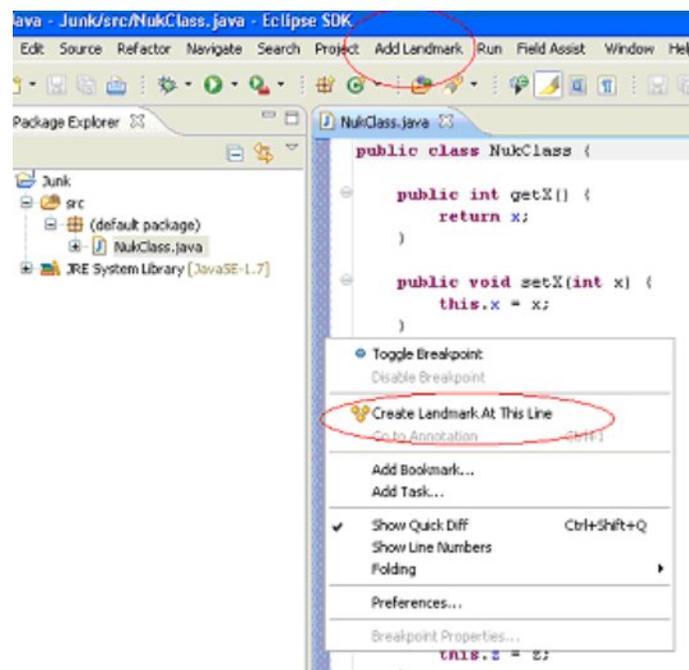


Figure 7: Creation of Landmarks from the main menu and decorator bar

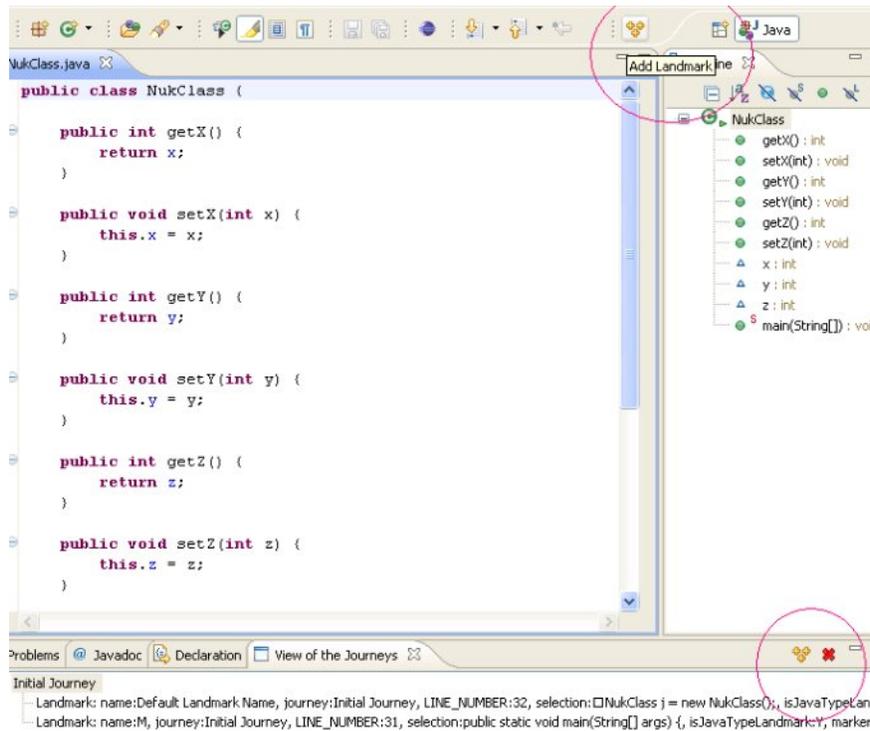


Figure 8: Creation of Landmarks from the main toolbar and Journeys view

4.8.2 Views

The feedback indicated that the Eclipse environment is complicated enough and, except for the specialized view, no additional visual elements were desired to add to the interface except for the specialized view. Adding Landmarks in the view required a change of focus from the source code to the view so it was less user-friendly than adding Landmarks without leaving the active editor.

In conclusion adding of landmarks was acceptable only from the source editor or it's decorator bar.

We decided to allow both methods as one (decorator bar) allowed quickly adding a Landmark for a line and the editor method allowed for selecting a custom section of code.

Using a specialized view integrated well in the Eclipse philosophy and existing UI so we decided to create a Journeys View.

As the Journey execution history needs to support large drawing real estate, we decided to create a separate view for this feature.

4.8.3 IDE context

Eclipse uses pre-packaged collections of views called Perspectives. Most perspectives are built based on activity. For instance, a Java perspective incorporates most views for basic code production, while the “Team Synchronizing” for source code management.

As we introduce this novel approach as an integrated part of code production, we decided that existing perspectives (Like Java or Debugging) should be used rather than creating new Perspectives.

4.8.4 Customizations of Journey Executions

For starting the live Journeys, we needed to use Launch configurations. Creating and managing normal Launch configurations is pretty straightforward however live environments require often extra customization in terms of start-up arguments, environment variable etc.

The goal was not to require developers to learn another way of managing the launch configurations so we decided to reuse the built-in facilities. Moreover, we decided to re-use the same launch configurations as for normal run and debug sessions. That way when we execute run Journeys the default launch configuration will be used or the user will be prompted for optional configurations should they exist.

4.8.5 Blending Journeys

During the launch of a journey, the system will stop at the prescribed Landmarks locations and display the associated documentation. The development environment also permits stopping at other breakpoint. That introduced the question of how to blend normal breakpoint stops with Landmark stops. There was the option of deleting all previous breakpoint at the start of the Journey or allow merger of the breakpoint.

The first iteration implemented a basic behavior where previous breakpoint- stops were disabled and new landmark stops were created. During the test trials developers suggested that in some cases they wanted to preserve their existing stops. The also suggested that in at least one case they wished to combine journeys. Developer L pointed out that in one case a product creation feature was used with an image-format generation feature. Performing a union between the two sets of Landmarks could have been useful in some cases.

As the result of the feedback, we decided to create several options so we can accommodate these preferences.

4.8.5.1 One journey

Firstly, create a simple to understand and use ‘Start Journey’ facility that would preserve all the stops produced by other means than other Journeys.

4.8.5.2 Non verbose running

Secondly, we created a ‘Map Journey’ facility that will create the selected Journey stops, disable the other Journeys’ stops and preserve the other types of stops. This will allow a developer to disable and re-enable selectively the stops. In this case, the running is triggered by using the normal Run Application Eclipse facility. In this mode, the Landmarks are visited but the landmark information is not presented.

4.8.5.3 Merge mode

Thirdly, we decided to add a ‘Map Journey in Merge Mode’ where new stops are added as a union with previous stops. This specific feature will permit running merged Journeys as per the scenario described by developer L. To allow the reset of the stops we also introduced a Un-map Journeys that removed all the Journeys created stops.

These is a unique feature, as none of the previous approaches allowed customized merging of multiple collection elements neither selective disabling of landmarks.

4.8.6 Landmark granulation level

Some of the previous work does not allow setting locations at the line level.

(Ko, Myers et al. 2006) determined that single line granulation is very common.

Many programming languages allow many to virtually unlimited statements concatenated on the same line.

In order to accommodate this reality we felt that we needed to support a block granulation like the blue selection in Figure 9: Creating landmarks for blocks of code.

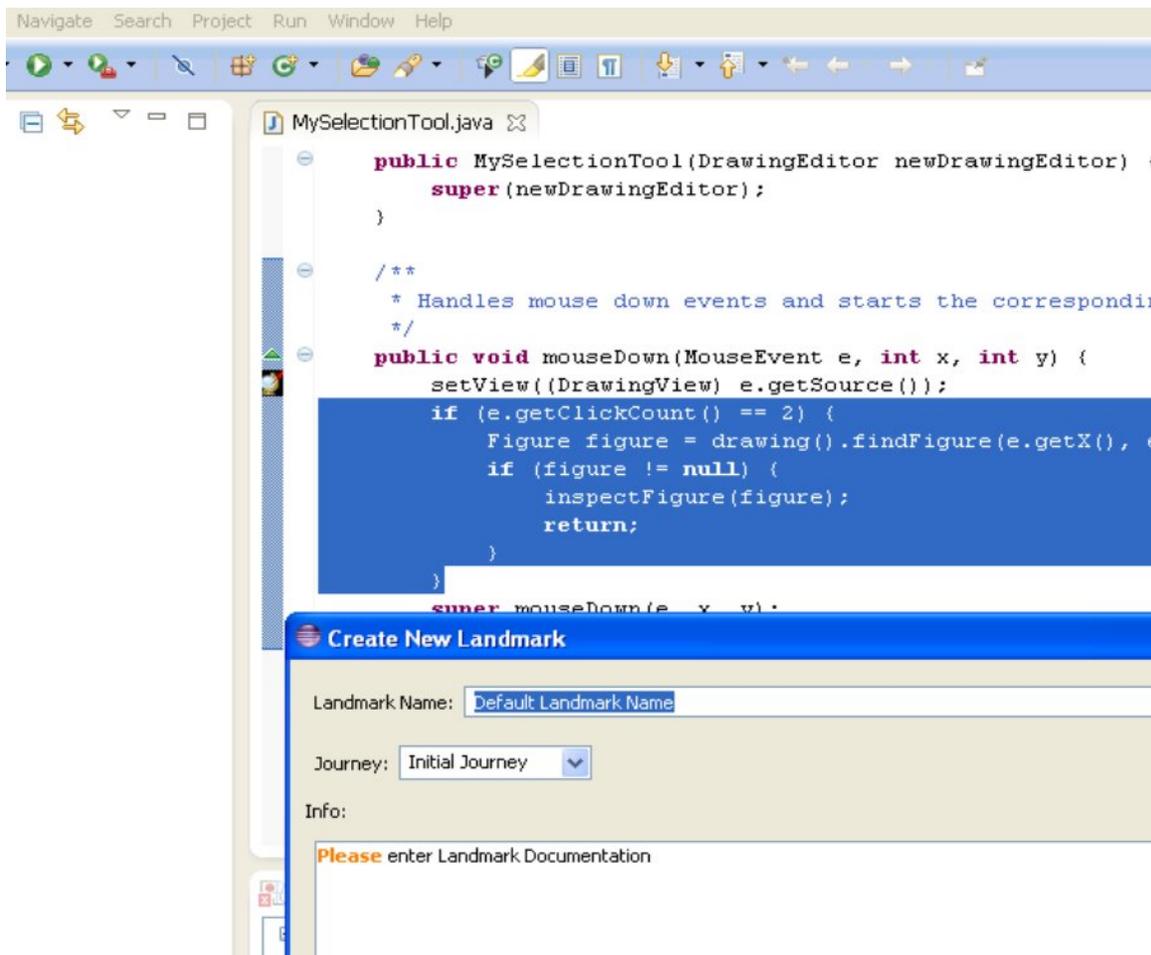


Figure 9: Creating landmarks for blocks of code

This decision is also supported by (Ko, Myers et al. 2006) that states: “*Most of the developers’ relevant information were single statements or pairs of statements*”

4.8.7 Landmark visitation constraints

In launch mode, one breakpoint is created for each Landmark. As normally Landmarks are visited only one per journey, we set the breakpoint properties/hit count to 1. We envision that in some cases we might require more visits for the same Landmark. In this case, we might also want to limit the number of visits by adding a Hit Count attribute. This additional optional attribute needs to be added via the Add New Landmark UI like in Figure 10: Restricting the number of visits for a landmark.. This feature is not currently implemented.

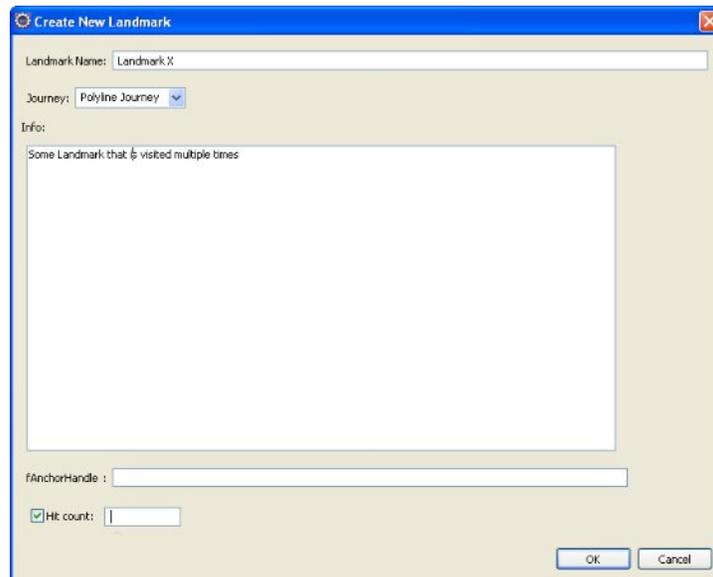


Figure 10: Restricting the number of visits for a landmark.

4.9 Implementation

The software was implemented as standard Eclipse plug-in. One of the challenges consisted in the “comprehension” of the Eclipse Framework and libraries.

Eclipse, as an environment is meant to be reused, so we utilized a significant number of libraries.

Graphic libraries, markers, menus, views and persistence were meant to configure, extend, and reuse.

On the other hand launching and debugging operations are quite intimate as part of the core packages. Their extensions were discouraged and created challenges.

A supplementary challenge arose from the heavy threading model that occasionally creates temporary de-synchronization.

For depicting history runs, we used an optional package from EMF (Eclipse Modelling Framework). That supports graph creation and even a special .dot format.

Chapter 5

Experiment

We reused the JTourBus experiment with a group of eleven experienced participants. In summary, the results show that people with the tool worked faster and got the correct answers more frequently.

5.1 Experiment Selection

Some of the literature in this research area proposes software solutions. Some of those proposed solutions are accompanied by experiments. Most of the experiments aim at assessing the increase of productivity by using the proposed method and sometimes they measure the productivity change due exclusively to the use of the tool.

The normal experiment design presents one or more software tasks to participating developers and measures the difference in speed in achieving the tasks. Most of the prior experiments use available open-source third party sample software. Some experiments are executed using existing software while other develop additional tools while some

We feel that reusing some of the previous experiment framework would add value to our work so we decided to reuse the JTourBus experiment. that uses the JHotDraw as experiment software.

For successfully finalizing a software task, participants must locate parts of the software relevant to the required change.

The experiment aims primarily to assess if productivity gains are made by using this approach.

The other aim of the experiment is to analyze how developers utilize the novel tooling, how useful they found this approach and what are their suggestions. These observations could be used to formulate new designs.

5.2 Sample Software used by the Experiment

The tasks given to the participants in the experiment require understanding of particular sections of some features present in the sample software. The sample software, JHotDraw (version 5.4) is a graphical framework originally developed for academic purposes by Erich Gamma, Thomas Eggenschwiler and W Kaiser to demonstrate object orientated design patterns.

JHotDraw contains framework libraries and several sample applications that utilize and demonstrate the framework. The particular executable application chosen for the experiment is a standalone Java application named JavaDrawApp. JavaDrawApp is a simple, vector-based graphical editor with built-in support for a number of basic geometric figures and basic graphic editing.

JHotDraw framework and JavaDrawApp sample application amount to 27 KLOC.

The original source code was adjusted for the purpose of this experiment.

We slightly simplified it by removing the references to the other sample applications, and formatted some of the source code to align it to current practices.

We also had to change several imports because name collisions between the current Abstract Window Toolkit (awt) libraries and some class names in the JHotDraw framework.

We made one code change consisting in closing an opened resource. That was necessary to avoid memory leaks and avoid system freezes.

5.3 Documentation used by the Experiment

JHotDraw used in the experiment contains 26,895 lines of Java code (plus 33,791 lines of comments) in 448 classes. Additionally the package contains 69 images. The original JavaDoc was present both in the source code and in a dedicated directory containing about 900 files in standard html format.

The participants placed in the Software Journey group received the Journey documentation. In order to avoid bias we decided to employ the same documentation as the JTourBus experiment.

Some adjustment was necessary to reflect our usage of the software and to avoid some minor issues.

One change was to change the label date on the architecture document to reflect the actual date of the file.

Another change was to remove the class level comments. An example of class level documentation that was not included is shown in Figure 11: Example of class level documentation that was not added to Journeys.

```
/**
 * @JTourBusStop 1.1. Handler API Route - How and why to implement an own
 * handler, Handle - Interface that a handle has to implement:
 *
 * Handles are used to change a figure by direct manipulation in the user
 * interface. They are represented by small yellow circles inside the selected
 * figure. Depending on the type of figure, they can be used to rotate it or to
 * resize it for example.
 *
 * Handles know their owning figure and they provide methods to locate the
 * handle on the figure and to track changes.
 *-----
 * Instead of implementing this interface you should implement the abstract
 * class AbstractHandle.
 */
```

Figure 11: Example of class level documentation that was not added to Journeys

5.4 Participants

Our research target is mature teams functioning in level three or higher, mature development environments according to the Capability Maturity Model (Paulk, Curtis et al. 1993) definition. This is an important factor in selecting the participants in the experiment. There are numerous studies pointing out that experienced developers have a different behavior and are more effective. For instance, (Widowski 1986) compared novice and expert Pascal developers and found the experts were much better. The same conclusion is reached in (Wiedenbeck, Scholtz 1989) which found that that advanced experienced developers were better at determining program function than novice developers.

We recruited eleven voluntary developers with industry experience in OOP development. All but two developers had good-to-extensive (five or more years) Java development experience. The great majority of developers had intermediate to advanced development experience. We also recruited two less-than-intermediate experience participants to simulate a typical mature development team. These two had more than 6 months recent full-time Java development experience, so they would be considered junior in comparison with the rest of the participants but considered intermediate developers otherwise. We allocated the developers to the test groups randomly within their seniority groups.

Several prior studies considered and analyzed subsets of developers (Ko, Aung et al. 2005). Given that this research focuses on mature organizations using mature processes, it was important to avoid analyzing novice Java programmers because of studies pointing to the high variability of their knowledge and decision-making (Curtis 1981).

The developers we studied were eight males and three females in the 20 - 55 age range. They all had full time industry experience as software developers. All worked extensively with OOP and had at least six months Java development experience using Eclipse.

None of the participants had prior knowledge of JHotDraw beyond its existence and none of them had seen the source code or used derived JHotDraw libraries.

The participants completed the experiment in a various locations, mostly workplaces. That had the advantage of offering a more realistic context and minimized the time expenditure.

The participants were not given advanced information regarding the total number of participants, number of groups or the nature of their group.

The Eclipse environment is quite rich in functionality and requires significant time to master. Using any new tool requires a new effort so the net effect on productivity could be negative.

In order to reduce the impact of the effort to understand and use the new software we kept the interface as unobtrusive and simple as possible. We also employed a user interface consistent with the rest of Eclipse UI to minimize the novelty impact.

To further reduce this impact, we provided the participants in advance with copies of the software and asked them to get familiar with the new software prior to the experiment. We also encouraged them to use the software in their current activities should they find this appropriate.

We allocated the participants based on experience, by sorting them by overall experience, Java experience and Eclipse Experience. We used an adjusted round-robin criterion to create the two groups and we selected the larger group to test our method. We call the control group the ‘paper’ or ‘P’ group and the Software Journey group the ‘tool’ or ‘J’ group as per

Table 3: Participant experience. Note, that one of the developers that filled out the pre-experiment survey was not available for the experiment so he was removed from the list.

Participant ID	Years of development experience	Years of Java development experience	Years of Eclipse/RAD/RSA development experience	Group
1	5	4	3	J
2	5	5	5	J
3	8	5	8	P
4	10	2	0.5	P
5	10	10	8	J
6	12	9	7	J
7	16	0	0.5	P
8	20	5	4	P
9	25	1	2	J
10	28	16	9	J
11	29	8	6	P

Table 3: Participant experience

5.5 Preliminary Survey

The questions and answers to the preliminary survey can be seen in Table 4: Survey questions and Table 5: Survey answers.

- Q1 Does your development team have a development process?
- Q2 Have you developed any industrial size systems in excess of hundred of thousands of lines of code?
- Q3 Have you used JavaDoc in the process of developing or updating code?
- Q4 Have you used the debugger and breakpoints in the process of understanding, developing or updating code?
- Q5 Have you used any bookmarking mechanisms?
- Q6 Have you used Eclipse plug-ins that do not come standard?
- Q7 Have you used UML in the process of documenting, understanding, developing or updating code?
- Q8 Age range

Table 4: Survey questions

Answers choices where (1 - strongly disagree, 2 - disagree, 3 - neither agree or disagree, 4 - agree, 5- strongly agree.)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
1	s	No	4	5	2	1	4	19-29
	Ye	Ye						
2	s	s	5	5	3	3	5	30-49
	Ye	Ye						
3	s	s	3	5	3	5	1	30-49
	Ye	Ye						
4	s	s	5	5	3	5	5	19-29
	Ye	Ye						
5	s	s	3	5	4	5	3	30-49
	Ye	Ye						
6	s	s	1	5	4	1	3	30-49

7	No	Ye s	4	5	5	5	5	30-49
8	No	Ye s	1	5	5	4	5	50-64
9	s	No	4	4	4	3	3	50-64
10	Ye s	Ye s	3	3	3	3	4	30-49
11	Ye s	Ye s	5	5	5	5	5	50-64

Table 5: Survey answers

5.6 Results

Detailed results based on the setup described after this section are shown in

Table 6: Raw Data.

The header contains tasks 1 to 4. The rows are cyan for participants from group ‘Paper’ and yellow for the participants from the group ‘Journey’. The incorrect answers are in red. Some tasks marked with N/A were not answered due to time constraints or lack of completely understanding of the task.

Participant/Task	T1	T2	T3	T4	Group
P1	19	3	3	25	Journey
P2	50	30	15	50	Journey
P3	64	22	N/A	17	Paper

					r
P4	15	14	59	20	Pape r
P5	30	65	30	20	Journ ey
P6	4	6	6	23	Journ ey
P7	50	36	20	10	Pape r
P8	28	8	10	12	Pape r
P9	11	N/A	N/A	N/A	Journ ey
P10	3	5	6	17	Journ ey
P11	44	35	50	35	Pape r

Red denotes an incorrect answer

N/A denotes a task that was not answered

Table 6: Raw Data

The detailed graphic where the answers are sorted by time is depicted in Figure 12: Experiment detailed results. The incorrect answers have been removed from this graphic. The paper experiment results are depicted on the left and the Journeys experiment results are shown on the right. Timing results on the Y-axis are sorted in ascendant order, based on the time it took developers to find a correct solution. Participants with incorrect answers are positioned after the longest time per task. It can be noted that except for Task 4 it generally took longer time for the Control group (left) compared to the time it took the Tool group (right).

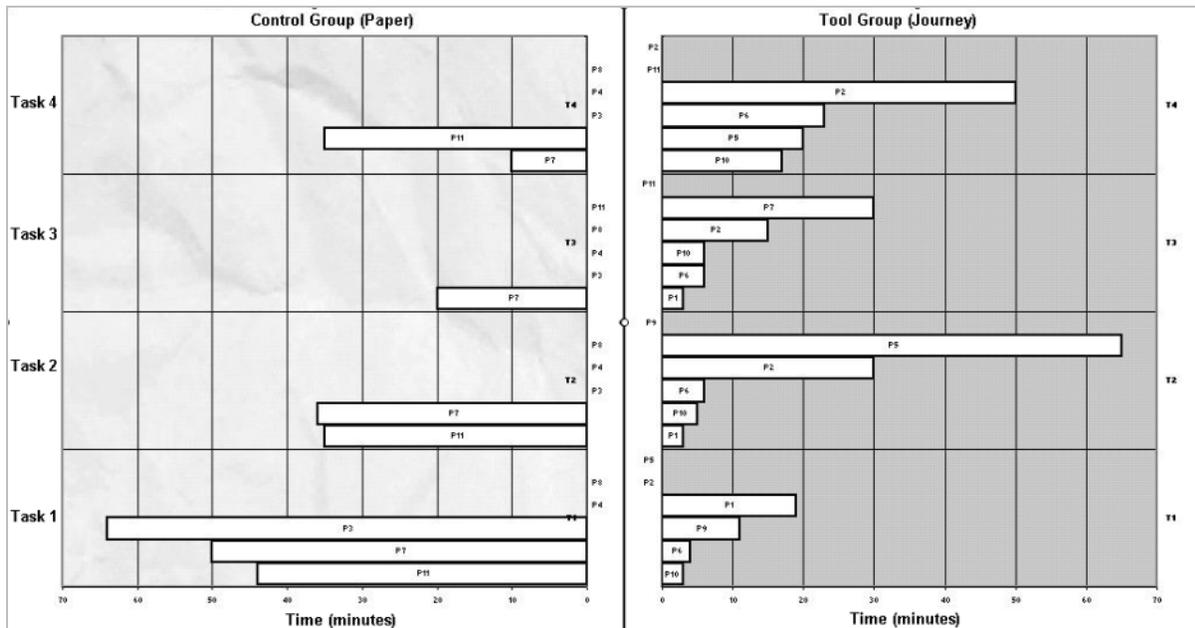


Figure 12: Experiment detailed results

Not all the candidates that filled out the participation survey ended up being available for the actual experiment however, the great majority (11 out of 12) ended up participating in the experiment. The results from the four tasks demonstrated that most participants found the correct solutions.

The last task has not provided a good differentiator as some participants have not attempted to solve it and the results of the ones that attempted was different from the original published solutions. As it was somehow hard to validate the answers as completely correct or incorrect, we decided not to include task five in the results.

There was some feedback on the Journeys tool itself concerning both the behavior and UI. P2 and P6 noted the overlapping tree elements in the Journeys execution view. They found quickly that enlarging the view window was providing a satisfactory solution.

Two of the participants (P10 and P1) noted the documentation pop-up at the beginning of the journeys. Participant P1 called the behavior ‘jarring’. Both reported becoming accustomed to this behavior later in the experiment. Participant P2 made remarks about the length of the documentation page. Subsequently the same participant found it acceptable once realizing that it is sorted based on content.

One participant (P6 in Journey group) performed searches using an external tool (Notepad++) during the experiment. The participant considered this setup as being superior in speed.

Task 4 had involved a geometrical figure creation and the created Journeys covered only post-creation editing of geometrical figures. One of the participants, (P2) started setting up new Landmarks in a trial and error manner to cope with this situation.

To better accommodate this we could introduce a complementary facility to create ‘Candidate /Hypothesis Landmarks’ that would be visible different from other Landmarks and could have a separate activation/deactivation mechanism.

Another participant (P5) “*Used existing landmarks and also set some new landmarks to get familiar with the code and identify the location of code that needs to be changed*” The same participant (P5) also “*added System.out.println at a few places to help me to finish the tasks*” It seems this participant used a classic method together with the new method. That might be an attempt to assess both these methods in parallel due to lack of trust or familiarity with the new method.

One participant separated the time it took to find the location and the time to validate the findings. For each of the four tasks, approximately half the time was spent on formulating a hypothesis and the other half on validating it.

The medians for the four tasks after removing all incorrect answers are depicted in Figure 13: Median time for task 1 to 4.

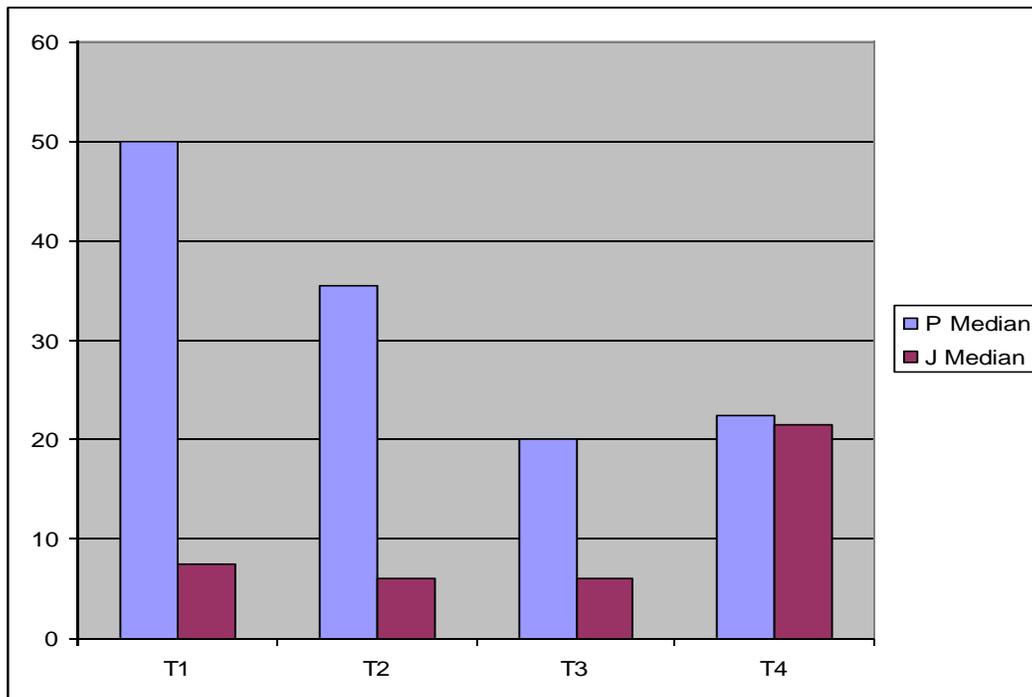


Figure 13: Median time for task 1 to 4

5.7 Experiment Set-up

The developers in our study were asked to complete tasks related to different software functionality.

The developers were given a maximum of 80 minutes per task before moving to the next task. Developers that finished all the tasks (according to the first rule) were told to return and continue on any unfinished tasks. That was aimed at maximizing the amount of completed work while avoiding developers spending too much on a task to the detriment of remaining tasks.

Two of the tasks were enhancement tasks and required developers to understand the system well enough to be able identify the location in the code where modifications needed to be made to achieve the goal.

The remaining tasks were related to defect fixing and required developers to test the program and identify a place in the code base where a fix was to be made.

To contain the duration of the experiment, developers were required to identify only one location out of potentially multiple locations.

To simplify the process and control for the software configuration factor we decided to use an Eclipse workspace archive that would be sent to the experiment participants in advance. The package for group J (the group using the Journeys software) contained also the Software Journeys plug-in.

The participants were directed to use the same software they use in their routine development practice and they were allowed to use the Internet. While we were inclined to control for additional tooling we believe that creating a close-to-reality environment would allow us to better assess the impact in productivity in realistic conditions.

The developers were given an empty text document and asked to capture details about any additional software before the start of the experiment.

The group of participants using the Journey approach received the associated documentation. In order to avoid bias we decided to employ the same documentation as the JTourBus experiment. The JTourBus documentation was created in two phases. According to the authors:

“In order to avoid bias, we did not produce the documents ourselves but rather employed three top students of a previous software engineering class. The first and second author wrote the J and P documentation together, to the best of their capabilities, but without knowing the task the subjects would be given” ... “A third author then reworked both of the resulting documents. He was told to modify both documents in such a way that they contained equivalent information and none would favor its users over the other”

As our approach was different, we had to move parts of the documentation from base classes in derived classes.

It is common to have several options of implementing required code changes. The chosen option can be influenced by process guidelines, coding standards or individual experience and style. Because some tasks had multiple valid solutions, we accepted as valid any viable solution.

5.7.1 Experiment specific instructions for group P (Paper)

We reused the JTourBus experiment instructions for group P.

5.7.2 Experiment specific instructions for group J (Journey)

The ‘Software Journeys’ group received additional instructions to introduce the tool:

You access this existing additional documentation by using a plug-in tool called Software Journeys.

This tool was used to capture feature documentation when software is developed.

The same tool can now be used to explore the captured documentation in order to get familiar with important locations in the code implementing a specific feature.

Capturing this documentation for each feature was deemed mandatory by the development process.

This extra documentation was already captured, so you will be using the tool just for viewing the previous documentation. You can use this facility in concert with any other tools and techniques you normally employ in your daily development activities.

This plug-in creates a new view called Journeys. The view contains several Journeys corresponding to a software feature like for instance an “Add bank Account” feature. Each Journey contains a collection of Landmarks. Landmarks are important code locations for a specific feature. Usually a Journey contains a Landmark located at the application starting code. For a Java application, that Landmark is located in the ‘main’ method. That initial landmark is a good place to start getting a sense of the lay of the land.

In order to execute the program and stop at these important landmarks you should right click on a Journey and select ‘Start Journey’.

To view Landmark documentation in static mode (the program is not running), right click a Landmark and select Properties.

By clicking on a Journey and selecting ‘Map Journey’, you can also ‘Map’ a journey. That creates breakpoints for the Landmarks in a Journey. Doing that, disables all the breakpoints associated with any other Journeys. You can run a Journey, firstly by creating breakpoints for Landmarks and then starting the

application. You can also merge several Journeys by mapping all selected Journeys using “Map Journey in Merge Mode” and then launching the application using built-in Eclipse debug mode facilities.

During the running of a Journey, you can activate or deactivate the Landmarks by enabling or disabling the corresponding breakpoints associated with other previously mapped Journeys.

The screenshot in Figure 14: Example of JHotDraw feature Journeys, contains a view of the Journeys showing documentation captured for three Journeys.

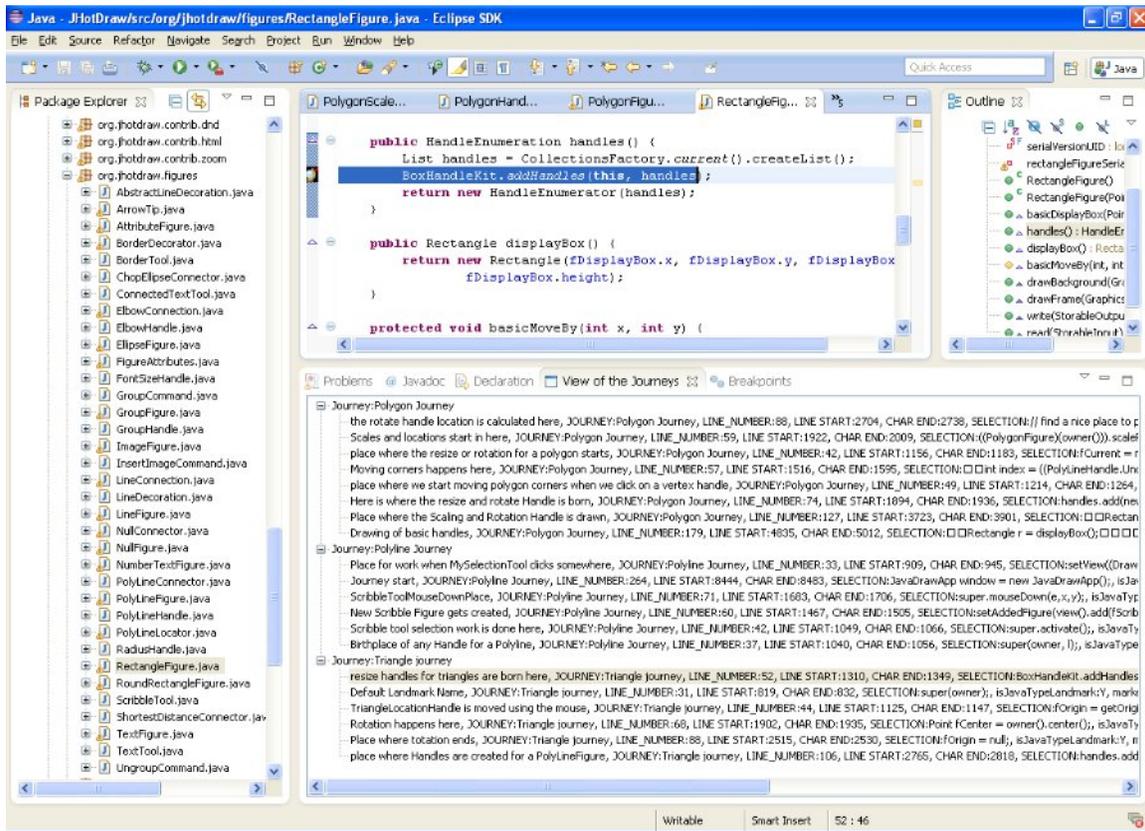


Figure 14: Example of JHotDraw feature Journeys

5.7.3 General information about the sample software

We reused the JTourBus experiment general information about the sample software.

5.7.4 Tasks

The tasks were identical to the one used by the JTourBus experiment except for a correction applied to Figure 16:Task 2 where we reversed the direction of the arrow to match the task.

Task 1

Using the ScribbleTool one can draw a connected series of line segments. At this point, such a series of line segments cannot be enlarged in the same way as for instance a triangle using the typical 8 resize-handles as per the figure. Please identify one of location in the code, which need to be modified in order to allow resizing using the same type of 8-resize handle as the triangle. One location is sufficient for this task. See Figure 15: Task 1

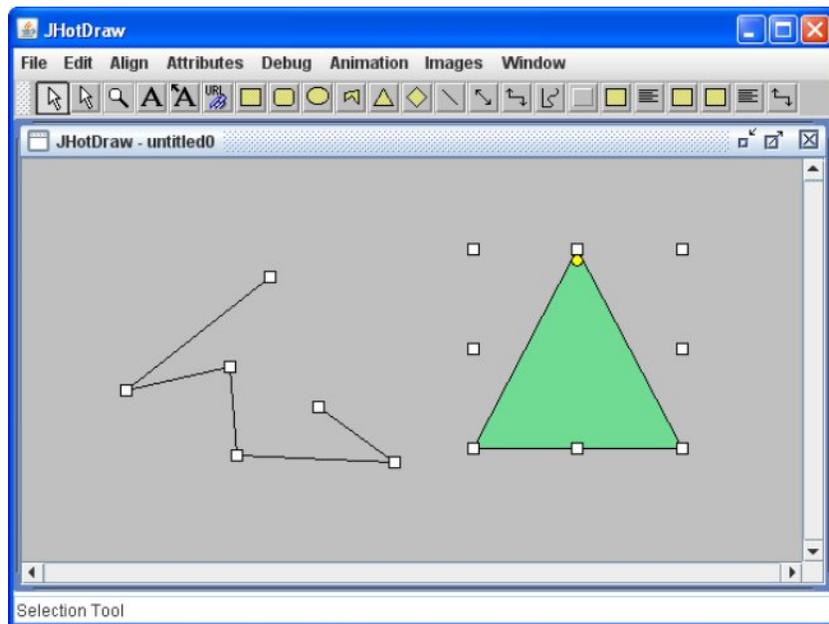


Figure 15: Task 1

Task 2

A new polygon has a little yellow handle for rotating the polygon that can appear in any corner. There is a bug in the current implementation, which makes the little yellow handle jump to another corner, when significantly moving any corner handles for changing the polygon. See Figure 16:Task 2

The following screenshot shows what is happening if the lower left corner is moved to the right:

Which location/s in the code need to be changed to avoid this behavior?

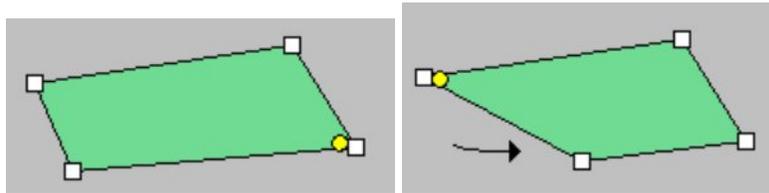


Figure 16:Task 2

Task 3

If several handles of the same figure are drawn on the same spot, the mouse always selects the one drawn at bottom. It would be preferred to always move the handle that is also drawn on top of the others. Shown below is an example situation where the handles overlap: See Figure 17: Task 3

Which location/s in the code need to be adapted, so that the handle on the top can be moved?

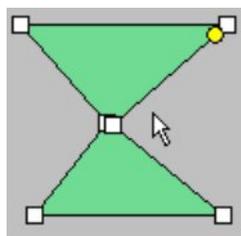


Figure 17: Task 3

Task 4

New figures can be inserted into an area marked by the mouse. From other graphic editors it is a well-known feature to insert new figures centered around the first click, if CTRL is being held. Which location in the code needs to be adapted, so that such a change is possible for ellipses?

Task 5

A last problem of JHotDraw is that you cannot abort a drag-action (for instance using escape). Which location in the code needs to be changed to enable this feature?

Chapter 6

Discussion

Software comprehension is a human activity and a one-size-fits-all approach is unlikely to be sufficient. There are a large number of factors that influence the state of an existing software system. Some systems are better documented, others followed a more consistent naming convention, some are more complicated in terms of relationships, some differ by the size of the code base etc.

Using searches in software developed using a consistent, well-known naming convention, might yield very good results for some parts of the system. Using landmarks with little or no documentation might be sufficient for certain tasks. The tool supports this as a light mode.

Other tasks might benefit from location-based documentation.

Some tasks might be solved using a static Landmark-documentation approach. The presented solution accommodates the static mode. In case the developers do not need system state information, call stack information, threading information, live expression evaluation the static approach might be employed.

In a way we can look at this tool features as supporting a gradual approach where a developer would just scan the locations, then continue by reading the documentation then continue by starting the journeys. The developer could stop at any of these stages should he/she be satisfied that the desired level of comprehension has been reached.

As this approach is not a magic bullet solution and there are cases where it might not work and some areas where it would not work.

- It might not work well for teams that do not develop based on a project plan because it would be harder to have a criterion for creating Journeys. Some open source projects might fall in that category.
- It would probably be unnecessary in cases where the source code is so small that one developer could memorize the important locations and their functionality.
- It would not be recommended for teams where not all developers use the same IDE in case that the other IDE does not support the Journey format and maintenance.

As with all systems subject to evolution, there is the danger of running out of synchronization between code and documentation. We believe that connecting the Landmarks directly to the code and having Landmarks visually visible for developers that maintain the code would be advantageous. In other systems, developers have to find the other documents in order to do updates. In that case, the connection might be considered weaker and the extra effort might discourage the developers from updating the documentation.

The choice of external persistence versus in-code persistence has advantages and disadvantages. In our approach, the documentation is internal to the source-control system but external to source code files. This is consistent with the way Eclipse treats persistence for breakpoints, bookmarks and other types of markers. Obviously, this does not work well if the source files are modified externally using non-IDE text editors as we have the danger of documentation location running out of synch. The current persistence model saves not only the document locations but also the marked content so out-of synch-detection is a matter of validating the saved content versus the existing content. That validation is not expensive, considering the limited number of Landmarks and the incremental nature of changing code.

This approach is geared toward comprehension and not towards regression testing. In consequence, system state values have to be used only for comprehension and not for testing. In some cases, expressions could offer a better comprehension hint and they are supported indirectly via breakpoint expressions. Not duplicating existing IDE functionality was a stated design principle, so new expression building features have not been added. These expressions would have to be associated to Landmarks (currently unimplemented feature).

Any particular concrete implementation invariably introduces some limitations. Eclipse has several limitations, mainly due to its multithreading environment where different operations execute on different threads. That allows a user to start new Journeys while existing Journeys are still ongoing. That could generate confusion and should not be attempted.

Similar to (Oezbek , Prechelt) experiment we note a sensible degree of difference from participant to participant in terms of productivity. Alternating tasks with and without the tool might have provided a better control. At the same time, that would have introduced a method switch between tasks and that might have introduced other factors difficult to control.

The proximity of the landmarks to the solution location for Task 1 did not seem to have a significant influence on the results. That is similar to the results from the JTourBus experiment.

The documentation covered the editing of already created geometrical figures. One might argue that the birthplace of figures would normally be considered a Landmark, As the experiment reused externally created tasks we wanted to stay as close to the original documentation and as such no creation time documentation has been added. Task 4 was not an edit type task but a creation task. One might speculate that that type of task would be equally difficult for both groups. Most participants in the Journey group noted this situation.

The fact that one participant added print statements could signify a personal preference or stage of acceptance of the tool. To better accommodate this we could introduce a complementary facility to generate marker print statements at Landmark locations. This can be done either at Journey or at Landmark level of granulation. Naturally, a cleanup facility should also be provided. Adding the print facility might duplicate the Journeys Execution View but would offer a more familiar format.

Task 1 was the task with the most incorrect answers and had some of the longest time durations. This is the task where the participants came first in contact with the sample software. Considering that, it could be reasonable to look at it as a training task. It is probable that knowledge acquired during solving of Task 1 was useful in solving some of the rest of the tasks.

The participant population involved in this experiment had between 5 to 28 years of experience. JTourBus participant experience has been described as “45 graduate students and 11 undergraduate students (3 unknown)”. We could venture to consider undergraduate and master students as mostly ‘novice’ developers. On the same token, we could consider the majority of participants in the Journey experiment as ‘expert’ developers.

The JTourBus experiment considered only the results for Task 1.

The ratio for correct /(incorrect or no-answer) for Task1 was 67% for the Journey/Tool experiment, versus 60% for the Journey/Paper experiment, versus 27% for the JTourBus/Tool experiment, versus 28% for the JTourBus/Paper experiment.

The median time for correct answers for Task1 was 7.5 minutes for the Journey/Tool experiment, versus 50 minutes for the Journey/Paper experiment, versus about 36 minutes for the JTourBus/Tool experiment, versus 35 minutes for the JTourBus/Paper experiment.

During the experiments, the ‘expert’ developers from the Journey/Tool experiment generally outperformed the ‘novice’ developers in terms of speed. That was expected.

When comparing the Journey/Paper experiment timing results with the JTourBus experiment (both tool and paper) we see that the ‘experts’ generally underperformed the ‘novice’ developers in terms of speed. That (roughly 25%) difference taken in isolation was not expected. A closer look shows that it was achieved at the cost of decreasing the correct answers from 60% to 28%. That shows that in some cases ‘novice’ developers can find correct answers slightly faster but with half the accuracy. That was not unusual.

Due to the scarcity of wrong answers given by ‘expert’ developers versus ‘novice’ developers, it would not be sensible to draw any conclusions regarding speed relations between the wrong-answers.

Chapter 7

Conclusion

We demonstrated that it is possible to extend previous static solutions that associate code locations with information and add a dynamic dimension.

We demonstrated that the tool implementation was possible and was supported by core features in the IDE.

While the experiment has not been performed on a statistically large population, it used mostly developers with significant industry experience. The experiment could be interpreted as pointing towards an increase of productivity for developers using the new approach.

7.1 Summary of Contributions

Creating a new type of feature-based documentation that has a dynamic dimension

Discussed and recommended a comprehensive approach that requires no extensive infrastructure change

Designed and implemented a new tool to support this new comprehensive approach

Performed an experiment where participants in the Journeys group had a lower number of incorrect answers and a faster overall response.

Performed post-experiment interviews with several available participants, captured and selectively discussed their experience.

7.2 Future Directions

It would be interesting to further research the potential of running combined Journeys

It would be interesting to add supplemental support for landmark level constraints

It might be beneficial researching how to represent historic runs in case of blended Journeys

It might be beneficial to augment the tool functionality with a feature where all source files belonging to a Journey could be opened in the editor.

Bibliography

- BROOKS, R., 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18, pp. 543-554.
- CHENG, L., DESMOND, M. and STOREY, M., , Tours - Presentations by Programmers for Programmers. Available: <http://tagsea.sourceforge.net/> [June, 2012].
- CHIKOFFSKY, E.J. and CROSS, J.H., 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Softw.*, 7(1), pp. 13-17.
- CORBI, T.A., 1989. Program understanding: challenge for the 1990's. *IBM Syst.J.*, 28(2),
- CURTIS, B., 1981. Substantiating programmer variability. *Proceedings of the IEEE*, 69(7), pp. 846.
- DELINE, R., KHELLA, A., CZERWINSKI, M. and ROBERTSON, G., 2005. Towards understanding programs through wear-based filtering, *Proceedings of the 2005 ACM symposium on Software visualization 2005*, ACM, pp. 183-192.
- ERNI, D.S., 2010. *Codemap -- Improving the Mental Model*
- FRONK, A., BRUCKHOFF, A. and KERN, M., 2006. 3D Visualisation of Code Structures in Java Software Systems, *ACM SoftVis Challenge 2006*.
- GAMMA, E., EGGENSCHWILER, T. and KAISER, W., , JHotDraw. Available: <http://www.jhotdraw.org> [October, 2012].
- GHABI, A. and EGYED, A., 2011. Observations on the connectedness between requirements-to-code traces and calling relationships for trace validation, *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering 2011*, IEEE Computer Society.
- GREEVY, O., LANZA, M. and WYSSEIER, C., 2006. Visualizing live software systems in 3D, *Proceedings of the 2006 ACM symposium on Software visualization 2006*, ACM, pp. 47-56.
- GUZZI, A., HATTORI, L., LANZA, M., PINZGER, M. and DEURSE, A., , Pollicino. Available: <http://www.st.ewi.tudelft.nl/~guzzi/pollicino/user-study-1/> [June, 2012].
- JOHNSON, K., , Kelly-Johnson [Homepage of Encyclopædia Britannica], [Online]. Available: <http://www.britannica.com/EBchecked/topic/305355/Kelly-Johnson> [June, 2012].

- KAELBLING, M.J., 1988. Programming languages should NOT have comment statements. *SIGPLAN Not.*, 23(10),.
- KIENLE, H.M. and MULLER, H.A., 2010. Rigi-An environment for software reverse engineering, exploration, visualization, and redocumentation. *Sci.Comput.Program.*, 75(4), pp. 247-263.
- KNUTH, D.E., 1984. Literate Programming. *The Computer Journal*, 27(2), pp. 97-111.
- KO, A.J., MYERS, B.A., COBLENZ, M.J. and AUNG, H.H., 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *Software Engineering, IEEE Transactions on*, 32(12), pp. 971-987.
- KO, A.J., AUNG, H.H. and MYERS, B.A., 2005. Design requirements for more flexible structured editors from a study of programmers' text editing, *CHI '05: CHI '05 extended abstracts on Human factors in computing systems 2005*, ACM, pp. 1557-1560.
- LATOZA, T.D., GARLAN, D., HERBSLEB, J.D. and MYERS, B.A., 2007. Program comprehension as fact finding, *ESEC-FSE '07: Proceedings of the 6th joint meeting of the European software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering 2007*, ACM Press, pp. 361-370.
- LATOZA, T.D., VENOLIA, G. and DELINE, R., 2006. Maintaining mental models: a study of developer work habits, *Proceedings of the 28th international conference on Software engineering 2006*, ACM, pp. 492-501.
- LIU, H. and LETHBRIDGE, T.C., 2002. Intelligent Search Methods for Software Maintenance. *Information Systems Frontiers*, 4(4), pp. 409-423.
- LORETAN, P., 2011. *Software Cartography, A Prototype for Thematic Software Maps*, University of Bern.
- OEZBEK, C. and PRECHELT, L., , JTourBus. Available:
<http://sourceforge.net/projects/jtourbus/> [June, 2012].
- OEZBEK, C. and PRECHELT, L., eds, 2007. *JTourBus: Simplifying Program Understanding by Documentation that Provides Tours Through the Source Code*.
- OLSEM, M.R., 1995. *Reengineering technology report*. 1. Software Technology Support Center.
- ORACLE, , JavaDocs [Homepage of Oracle Corp.], [Online]. Available:
<http://docs.oracle.com/javase/1.5.0/docs/api/index.html> [October, 2012].

PARNIN, C. and ORSO, A., 2011. Are automated debugging techniques actually helping programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis 2011*, ACM, pp. 199-209.

PAULK, M.C., CURTIS, B., CHRISISS, M.B. and WEBER, C.V., 1993. Capability maturity model, version 1.1. *Software, IEEE*, 10(4), pp. 18-27.

ROBILLARD, M.P., , Feat Tool. Available: <http://www.cs.mcgill.ca/~swevo/feat/> [June, 2012].

ROBILLARD, M.P. and MURPHY, G.C., 2007. Representing concerns in source code. *ACM Trans.Softw.Eng.Methodol.*, 16(1), pp. 3.

ROBILLARD, M.P. and WEIGAND-WARR, F., , ConcernMapper. Available: <http://www.cs.mcgill.ca/~martin/cm/> [June, 2012].

SANGAL, N., JORDAN, E., SINHA, V. and JACKSON, D., 2005. Using dependency models to manage complex software architecture. *SIGPLAN Not.*, 40(10), pp. 167-176.

SILLITO, J., MURPHY, G.C. and DE VOLDER, K., 2006. Questions programmers ask during software evolution tasks, *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering 2006*, ACM, pp. 23-34.

STOREY, M., ARANDA, J., DAMIAN, D., PETRE, M. and WILSON, G., 2011. *How do practitioners perceive software engineering research?* <http://margaretannestorey.wordpress.com/2011/05/20/how-do-practitioners-perceive-software-engineering-research/>: ICSE.

STOREY, M., CHENG, L., BULL, R. and RIGBY, P., , TagSea. Available: <http://tagsea.sourceforge.net/> [June, 2012].

STOREY, M.A., 2011-last update, ICSE 2011 Panel on “What Industry Wants from Research”. Available: annestorey.wordpress.com/2011/08/05/icse-2011-panel-on-%E2%80%9Cwhat-industry-wants-from-research%E2%80%9D/.

STOREY, M.A., RYALL, J., BULL, R.I., MYERS, D. and SINGER, J., 2008. TODO or to bug: exploring how task annotations play a role in the work practices of software developers, *ICSE '08: Proceedings of the 30th international conference on Software engineering 2008*, ACM, pp. 251-260.

STROULIA, E. and SYSTA TARJA, 2002. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Appl.Comput.Rev.*, 10(1), pp. 8-17.

- WIDOWSKI, D., 1986. Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices, *In H P Willumeit (Ed.) Human decision Making and Manual control*, S. 267 1986, North-Holland, Elsevier.
- WIEDENBECK, S. and SCHOLTZ, J., 1989. Beacons: a knowledge structure in program comprehension, *Proceedings of the third international conference on human-computer interaction on Designing and using human-computer interfaces and knowledge based systems (2nd ed.)* 1989, Elsevier Science Inc.
- WILDE, N. and CASEY, C., eds, 1996. *Early field experience with the Software Reconnaissance technique for program comprehension*.
- WU, J. and STOREY, M.D., 2000. A multi-perspective software visualization environment, *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research* 2000, IBM Press, pp. 15.
- YING, A.T.T. and TARR, P.L., 2007. Filtering out methods you wish you hadn't navigated, *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange* 2007, ACM, pp. 11-15.