UNIVERSITY OF CALGARY

Pragmatic Software Reuse

by

Reid Holmes

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

NOVEMBER, 2008

© Reid Holmes 2008

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Pragmatic Software Reuse" submitted by Reid Holmes in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dr. Robert J. Walker
Supervisor
Department of Computer Science

Dr. Jörg Denzinger
Department of Computer Science

Dr. Saul Greenberg
Department of Computer Science

Dr. Chad Saunders
Haskayne School of Business

Dr. Susan E. Sim
External Examiner
Department of Informatics
University of California, Irvine

Date

Abstract

Many software reuse tasks involve reusing source code that was not designed in a reusable manner. These *pragmatic* reuse tasks arise for a variety of organizational and technical reasons. To investigate a pragmatic reuse task, a developer must navigate through, and reason about, unfamiliar source code in order to identify those program elements that are relevant to his reuse task and to decide how they should be reused. Once these elements have been identified, the developer must convert his mental model of the task into a set of actions he can perform. These tasks are poorly supported by modern development tools and practices.

The thesis of this dissertation is that by providing developers with a mechanism to create pragmatic reuse plans in a structured way, and a methodology to semi-automatically perform the pragmatic reuse task using this plan, we can enable developers to perform pragmatic reuse tasks more quickly and with greater confidence.

To validate these claims we have created a model that captures the program elements involved in a pragmatic reuse task, along with annotations that correspond to the developer's decisions about how an element should be treated in a reuse task; the model explicitly enumerates each of the actions required to perform a pragmatic reuse task. We have created a tool, called Gilligan, that allows developers to create pragmatic reuse plans by navigating through unfamiliar source code and annotating its structural elements corresponding to how they should be treated in a reuse task. Using this plan, Gilligan can semi-automatically transform the source code from its originating system and integrate it into the developer's system.

We have evaluated Gilligan using a series of case studies and experiments using a variety of source systems and tasks. The results show that pragmatic reuse plans are a robust metaphor for capturing pragmatic reuse intent and that Gilligan can significantly decrease the amount of time developers require to perform pragmatic reuse tasks.

Table of Contents

Abstract . . . iii

List of Tables . . . ix

List of Figures . . . xi

Acknowledgments . . . xiii

Dedication . . . xv

1 Introduction . . . 1
- 1.1 Pragmatic reuse . . . 2
- 1.2 Why pragmatic reuse is difficult . . . 3
  - 1.2.1 How pragmatic reuse difficulties relate to task categories . . . 5
- 1.3 Enabling pragmatic reuse . . . 5
- 1.4 Thesis statement and contributions . . . 6
- 1.5 Organization . . . 7

2 Motivation . . . 9
- 2.1 Alternative to pragmatic reuse . . . 16
- 2.2 Summary . . . 17

3 Related Work . . . 18
- 3.1 Software reuse . . . 18
  - 3.1.1 Black-box reuse . . . 19
  - 3.1.2 White-box reuse . . . 20
  - 3.1.3 Code clones . . . 21
- 3.2 Program understanding . . . 21
- 3.3 Program transformation . . . 22
- 3.4 Software visualization . . . 23
- 3.5 Cognitive aspects of reuse . . . 23
- 3.6 Summary . . . 24

4 Industrial Applicability Survey 25
4.1 Survey Results 27
4.2 Other Findings 29
4.2.1 Selected Questionnaire Responses 30
4.3 Survey Limitations 32
4.4 Summary 32

5 Pragmatic reuse model 33
5.1 Model requirements 33
5.2 Concrete model implementation 35
5.2.1 Relationships that exist between structural elements 35
5.2.2 Decisions a developer can make about a structural element 36
5.3 Pragmatic reuse plans as lightweight specifications 37
5.4 Relationship to the concern graph model 38
5.5 Structural analysis 38
5.6 Summary 39

6 Graph-based pragmatic reuse planning 40
6.1 Design goals 40
6.2 First Gilligan prototype 41
6.2.1 Using the graph-based Gilligan prototype 48
6.2.2 Evaluating the reuse plan 48
6.2.3 Application to motivational scenario 48
6.3 Evaluation 49
6.3.1 Case study 1 50
6.3.2 Case study 2 51
6.3.3 Case study 3 52
6.3.4 Case study 4 52
6.4 Discussion 53
6.4.1 Limitations 54
6.5 Summary 54

7 Tree-based pragmatic reuse planning 56
7.1 Design goals 56
7.2 Second Gilligan prototype 58
7.2.1 Application to the motivational scenario 62
7.3 Evaluation 62
7.3.1 Experimental tasks 64
7.3.2 Results 65

7.3.3 Observations 66
7.4 Discussion 67
7.4.1 Limitations 67
7.5 Summary 68

8 Pragmatic reuse plan enactment 73
8.1 Design goals 74
8.2 Third Gilligan prototype 75
8.2.1 Extraction 76
8.2.2 Integration 77
8.2.3 Supporting iterative planning and enactment 81
8.3 Evaluation 82
8.3.1 Task descriptions 83
8.3.2 Analysis of minimum required effort 83
8.3.3 Task effectiveness experiment 85
8.4 Discussion 87
8.4.1 Limitations 88
8.5 Summary 90

9 Holistic evaluation 93
9.1 Hypotheses 94
9.2 Participants 95
9.3 Tasks 95
9.3.1 Phase 1 96
9.3.2 Phase 2 98
9.4 Experimental procedure 99
9.4.1 Performing a trial 100
9.4.2 Data collection 100
9.5 Experimental design 101
9.6 Quantitative results 103
9.6.1 **H-1** analysis 103
9.6.2 **H-2** analysis 104
9.6.3 **H-3** analysis 105
9.6.4 Other quantitative analyses 107
9.7 Qualitative results 107
9.7.1 Organic concept categories 108
9.7.2 Interesting themes and quotes 113
9.7.3 Prompted concept categories 114

9.8 Discussion . . . 116
9.8.1 Characterization of good tasks as bad ones . . . 116
9.8.2 Giving up on bad tasks . . . 116
9.8.3 Gilligan improvements . . . 117
9.8.4 Limitations . . . 118
9.9 Summary . . . 118

10 Discussion . . . 120
10.1 Alternative reuse strategies . . . 120
10.2 Limitations of approach . . . 121
10.3 Evaluation . . . 121
10.4 Future work . . . 122

11 Conclusion . . . 124
11.1 Contributions . . . 125

Bibliography . . . 126

A Industrial Pragmatic Reuse Survey . . . 134
A.1 Industrial Pragmatic Reuse Questionnaire . . . 135
A.2 Industrial Pragmatic Reuse Case Study Questionnaire . . . 146

B Sample Graph Layout Algorithms . . . 148

C Planning evaluation . . . 152
C.1 Planning Case Study Instructions . . . 153

D Visualization evaluation . . . 157

E Enactment evaluation . . . 166
E.1 Effort . . . 166
E.1.1 Extracting the Metrics Lines-of-Code Calculator . . . 166
E.1.2 Extracting the Azureus Network Throughput View . . . 168
E.2 Enactment Experiment Instructions . . . 172

F Full evaluation . . . 180
F.1 QIF test file . . . 185
F.2 Entrance questionnaire . . . 187
F.3 Mid task questionnaire . . . 189
F.4 Post task questionnaire . . . 191
F.5 Exit questionnaire . . . 193

G Final Experiment: Card Sort Data 195
G.1 Organic Concept Categories 196
G.1.1 Dependency Identification 201
G.1.2 Understanding 209
G.1.3 Mental Models 213
G.1.4 Hypothesis Testing 219
G.1.5 Performing Pragmatic Reuse Tasks 223
G.2 Prompted Concept Categories 227
G.2.1 Pragmatic Reuse: Rationale, Impediments, and Frequency 227
G.2.2 Gilligan: Suggestions for Improvement 229
G.2.3 Observations 232

H Calgary Research Ethics Board Approval 237
H.1 Ethics Approval 5005 238
H.2 Ethics Approval 5605 239

List of Tables

2.1 Scope of the Azureus network throughput graph reuse task. 14

4.1 Likert-scale questions from industrial questionnaire. 27
4.2 Selected industrial pragmatic reuse survey results. 29

6.1 Standard Eclipse icons used by Gilligan. 43
6.2 Kinds of edges used by the Gilligan graph-based representation. 43
6.3 Gilligan colour scheme for encoding pragmatic reuse intent. 44

7.1 Icon overlays used by Gilligan to show additional information. 62
7.2 Size of the correct solutions. 64

8.1 Compilation errors for each task and treatment. 84
8.2 “Edits” required for each task and treatment. 85

9.1 Overview of participant experience. 96
9.2 Four blocks for the first phase of the experiment. 102
9.3 Contingency table for success and failure compared to treatment. 105
9.4 Overview of the organic concept categories. 112
9.5 Frequency that participants perform pragmatic reuse tasks 115
9.6 Participant’s rationale for performing pragmatic reuse tasks 115
9.7 Impediments to pragmatic reuse tasks identified by participants 115
9.8 Suggested improvements for Gilligan 116
9.9 Gilligan user interface shortcomings 116
9.10 # of participants who felt the task was good or bad. 117

A.1 Pragmatic reuse industrial survey responses. 144
A.2 Pragmatic reuse industrial survey aggregate responses. 145

D.1 Answer key for assessing precision and recall for Task 1. 158
D.2 Answer key for assessing precision and recall for Task 2. 160
D.3 Answer key for assessing precision and recall for Task 3 161

D.4 Answer key for assessing precision and recall for Task 4 164
D.5 Data table for second prototype evaluation 165

G.1 Overview of the organic concept categories. 200
G.2 Frequency that participants perform pragmatic reuse tasks. 228
G.3 Participant's rationale for performing pragmatic reuse tasks. 229
G.4 Impediments to pragmatic reuse tasks identified by participants. 229
G.5 Suggested improvements for Gilligan. 231
G.6 Gilligan user interface shortcomings. 231
G.7 # of participants who felt the task was good or bad. 234
G.8 # of participants who felt the scale of dependencies was problematic. 235

List of Figures

2.1 The UltiGPX application 10
2.2 Mockup submitted by the bug submitter 11
2.3 Network throughput view in the Azureus BitTorrent client 12
2.4 Mockup of UltiGPX reusing the Azureus view 13
2.5 Simplified class diagram for the Azureus reuse task 14
2.6 UltiGPX successfully reusing the Azureus view 16

6.1 Gilligan's role in the pragmatic reuse process. 42
6.2 Screen capture of the graph-based Gilligan prototype 47
6.3 Azureus graphics feature reuse plan. 47

7.1 Initial Gilligan dialog for starting a pragmatic reuse task. 59
7.2 Gilligan dialog for selecting the initial starting point. 60
7.3 Initial view presented by the second Gilligan prototype. 60
7.4 Expanded screen capture of a Gilligan Structural View 69
7.5 Screen captures demonstrating Gilligan's selection behaviour. 70
7.6 Gilligan source code view before and after decisions were made. 71
7.7 Screen capture while working on the motivational scenario. 71
7.8 Recall for the four tasks. 72
7.9 Average recall and time. 72

8.1 Gilligan's role in the pragmatic reuse process. 76
8.2 Example of how Gilligan rejects structural elements. 79
8.3 Screen captures of the third Gilligan prototype 91
8.4 Updated problem view after accepting `COConfigurationManager` 92
8.5 Snippet to resolve mismatch in Metrics task. 92
8.6 Productivity differences afforded by Gilligan's semi-automation 92

9.1 Block B1 time to completion by treatment 104
9.2 Block B2 time to completion by treatment 105
9.3 Peak open editors by task–treatment 106

B.1 Tree layout. 148
B.2 Tree layout enforcing a depth hierarchy. 149
B.3 Box-style tree layout. 150
B.4 Radial layout. 151

E.1 Gilligan enactment error for nested elements 168
E.2 Error caused by Gilligan not considering field ordering. 169
E.3 Pragmatic reuse plan for the Metrics task. 170
E.4 Pragmatic reuse plan for the Azureus task. 171

F.1 Pragmatic reuse plan for the QIF Parser task. 181
F.2 Pragmatic reuse plan for the Related Artists task. 182
F.3 Test harness for the QIF parser task. 183
F.4 Test harness for the related artists task. 183
F.5 Test harness for the torrent downloader task. 184

Acknowledgments

Graduate school has been an unexpected and exciting journey for me. For eight months in the third year of my undergrad I was sucked into the land of the SPL at UBC under the guidance of Gail Murphy. Before this point I hadn't really given much thought to grad school or what research was all about, but Gail and the SPL crew gave me a fantastic sense of what the academic lifestyle was like. Without Gail's early encouragement I would not be here today, and my life would be less full for it. Thank-you. The SPL was also full of great people who always knew how to help. Martin, Jonathan, and John, thanks for all of your advice. Brian, I have been seeking your advice since I was an undergrad and I will continue to do so in the future; thanks for all of your council.

After finishing my MSc with Gail at UBC, the next stop on this crazy train was at U of C with Rob Walker. Rob pitied the foo (that's me), obviously deciding I would be pliant enough for him to experiment on as his first PhD student. Being Rob's first "subject" was a fantastic experience; whenever I needed some advice, direction, encouragement, or a kick in the pants, Rob was there, keeping me on track, grounded, and focused. Rob taught me a great deal about how to design and perform an effective research program, and the importance of understanding the big picture; I aim to build on these lessons for the remainder of my career. I will miss randomly dropping by Rob's office for hours of conversation most of all.

ICT 524: I have outlasted all of you. Mark, Bhavya, and Punneet, thanks for the camaraderie. Kevin, you sure know how to make industry look good in a short period of time. Rylan and Brad, you guys continually surprise me by finding new ways to jump even further beyond the line of good taste than your previous foray (you were pretty good at reviewing papers too). Tom, thanks for all your advice; whenever I need the euro-perspective, you're the guy to get it from. To all in the LSMR, I look forward to reading many great papers in the future. Keep them coming.

Mark, I'm glad you made the trip out from UBC at the same time I did, it was nice to have some continuity in the department. Andrea and Mark deserve special recognition for helping me make the penultimate evaluation of this dissertation cogent. Morgan and Dan: thanks for the good times climbing, scrambling and at the games table; I'm not done with the Rockies yet, you'll be hearing from me for a while to come. Chris, I will never forget that you convinced me to start telemarking and then promptly abandoned it; this document must hurt you more than most: you won't be able to ridicule my student status any longer.

I would be remiss not to mention two specific troublemakers who convinced me that climbing 55 days outside was the normal grad student summer quota. To Steph and Brown: many good pitches were climbed, turns made, and peaks bagged. Three trips stand out to me: deciding that my first time on touring skis ever should be the Columbia Icefields in February, heading into the Bugaboos for the first time without having even heard of them or knowing what they really were, and finally the trip to the Tantalus where everything just went right. I know we all have some lists that will need ticking over the next decade or two.

Most of all, I would like to thank my family for being so supportive over the last 6.5(!) years. Laura, without your support this journey would have been meaningless; thanks for always helping me up when I've been down, giving me perspective, and most of all for your love. We've moved through four cities so far, lets hope we only have to do it one more time. Zoe, you might not really know what has been going on for the past 512 days, but your smiles always made my time at work a little more fun, and thinking of your big hugs made me bike home faster each night. Dawn and Brett, thanks for always helping us with Zoe when the crunch was on. To my parents and brothers, thanks for all the support. I can rest easy now, knowing that I will be the last Holmes boy to get a real job.

As this document technically concludes the 24 years I have spent on the pupil side of the educational equation, I will be reluctant to give up using 'but I'm just a student' as an excuse for anything I don't want to do, or pay for; however, I will steadfastly refuse any effort to get me to wear pants of appropriate length, stop evaluating the value of any foodstuff in terms of calories per dollar, replace the one bike shirt I have worn daily for the past 8 years, or make me feel self-conscious while stuffing free food into my pockets at every opportunity.

Reid Holmes

University of Calgary
December, 2008

To my ladies: Laura for getting me started, and Zoe for getting me finished.

Zoe Mae Holmes

A busy day on the NE Face of Mount Garibaldi

Zoe and Laura

Chapter 1

Introduction

Software reuse approaches encourage the development of software systems using pre-existing artifacts instead of creating them from scratch [Krueger, 1992]. Reuse has long been advocated as a mechanism to reduce development time, to increase developer productivity, and to decrease defect density [Mcilroy, 1968; Standish, 1984; Brooks, 1987; Krueger, 1992; Poulin et al., 1993; Boehm, 1999]. Subsequent studies have been performed to provide initial validation of these claims [Basili et al., 1996; Succi et al., 2001; Frakes and Succi, 2001; Ajila and Wu, 2007]. While several classifications of reuse approaches have been compiled [Krueger, 1992; Prieto-Díaz, 1993; Rothenberger et al., 2003], the majority of software reuse research has focused on black-box reuse approaches, such as software components [Mcilroy, 1968; Meyer, 1999; Szyperski, 2002; Ravichandran and Rothenberger, 2003], and program families or software product lines [Dijkstra, 1972; Parnas, 1976; Krueger, 2000].

Both reusable component approaches and software product lines suffer from three main drawbacks: (1) predicting the future— to know which pieces of software should be built in a reusable fashion— is notoriously difficult in practice [Tracz, 1990; Gaffney and Cruickshank, 1992; van Gurp and Bosch, 2002]; (2) developing all software in a reusable fashion is economically infeasible [Gaffney and Cruickshank, 1992; Cordy, 2003]; and (3) even software designed in a reusable fashion embeds a set of assumptions about how it is to be reused that can hamper its ability to be deployed in many contexts [Biggerstaff, 1994; Garlan et al., 1995].

In contrast to the controlled reuse scenarios with which reusable components and software product lines are associated, developers often find themselves in a position where a development task they are performing is familiar to them: either they have implemented the functionality in the past that their task requires or they have access to existing source code that provides the functionality. As reusable code is expensive to create, it is likely that the source code they are interested in reusing is not designed in a way that facilitates black-box reuse [Garlan et al., 1995].

In these situations, the developer has three main options: (1) to re-implement the functionality; (2) to refactor the original source code; or (3) to reuse the source code in an ad hoc, copy-and-

modify manner. There are numerous drawbacks to each of these options. Re-implementing the functionality is expensive and does not leverage any testing or other positive attributes associated with mature source code. Refactoring the original source code is often not feasible for a number of organizational and technical reasons: the developer may not own the existing code, the existing code may already be deployed and cannot be modified, the developer may be unwilling to introduce defects into the system the code resides in by refactoring something that is known to work already, and the developer may be unwilling to accept the security implications associated with shared source code. Reusing source code in a copy-and-modify approach can cause the developer to make poor decisions and the lack of tool support for these tasks makes them error-prone [Garnett and Mariani, 1990; Krueger, 1992]; however, this latter type of reuse is often the *pragmatic* choice in industrial scenarios.

1.1 Pragmatic reuse

> "To write a really good invisible avatar from scratch would take a long time, but he puts one together in several hours by recycling bits and pieces of old projects left behind in his computer. Which is how hackers usually do it."
>
> Hiro Protagonist [Stephenson, 1992, Chapter 48]

Reusing source code that was not designed in a reusable fashion has been known by many monikers: code scavenging [Krueger, 1992], ad hoc reuse [Prieto-Díaz, 1993], opportunistic reuse [Rosson and Carroll, 1993], and copy-and-paste reuse[1] [Lange and Moher, 1989]. We choose to introduce the term *pragmatic reuse* because each of these previous labels are associated with negative connotations, while we believe that pragmatic reuse tasks can be appropriate and effective. Indeed, non-black-box reuse tasks are not atypical; a recent study found that 47% of the reused source code within NASA, an organization committed to reuse, was reused in a non-black-box manner [Selby, 2005].

While research has stated that pragmatic reuse can be effective [Krueger, 1992; Frakes and Fox, 1995], little research has been performed to identify how industrial developers reason about and perform these tasks. Krueger states that, "In practice, the overall effectiveness of [pragmatic software reuse] is severely restricted by its informality" [Krueger, 1992]. Frakes and Kang [2005] identify two impediments to pragmatic reuse tasks: first, they state that development tools may not be effective at promoting reuse; second, they note that the lack of process associated with these hampers reuse efforts. Other researchers have identified similar shortcomings [Krueger, 1992; Sen, 1997; Morisio et al., 2002; Ravichandran and Rothenberger, 2003]. While these two impediments (informality and lack of tool support) may not constitute all of the shortcomings of pragmatic reuse approaches, they are the drawbacks most prominently mentioned in the literature. This dissertation investigates each

[1] Also known as cut-and-paste or copy-and-modify.

of these impediments by creating a model of pragmatic reuse tasks and accompanying tool support to make their performance more systematic, less error prone, and less time consuming.

Krueger [1992] states that reuse takes place in three stages: (1) the entity to be reused is *selected*; (2) the entity is *adapted* to fit its context of reuse; and finally (3) the entity is *integrated* into the target system. The selection portion of a pragmatic reuse task is generally undertaken informally: the developer knows of some appropriate source code they can reuse, or learn about it from their colleagues; while alternatives exist, we do not further address selection within this dissertation. Once the developer has identified a system containing the functionality they want to reuse, they must then delineate those portions of the source that are relevant to their functionality of interest from those that are not. During adaptation the relevant source code is removed from its originating system while integration is the process of inserting the reused code within the developer's system. These three steps can be overwhelming and complex for large pragmatic reuse tasks as they involve navigating through, and reasoning about, large amounts of unfamiliar source code.

Before any source code can be reused pragmatically, the extent to which the functionality is spread throughout its system, and the scope of its dependencies upon the system, must be understood [Garlan et al., 1995]. Understanding the structural dependencies—that is how the functionality calls, references, and is involved in the type hierarchies of its system—is central to assessing the suitability of a pragmatic reuse task. If the functionality is self-contained and only references other structural elements that are relevant or complement the developer's system, the reuse task will likely be easy. Alternatively, if the functionality is tightly coupled with large swaths of the originating system, it may be difficult to only reuse those relevant portions of the source code. Understanding these dependencies is essential to making an accurate assessment of the difficulty, and suitability, of a pragmatic reuse task.

1.2 Why pragmatic reuse is difficult

Pragmatic reuse tasks require the developer to navigate through a large amount of unfamiliar code, understand how it works, interpret how it relates to their task, determine which mechanical steps are necessary to perform the task, and then actually perform those steps. Upon observing developers performing pragmatic reuse tasks (as described in Chapter 9), we identified five conceptual categories that are pertinent to developers performing pragmatic reuse tasks. Many aspects of our approach will be related to these conceptual categories.

1. *Identifying structural dependencies:* Locating all of the relevant structural elements, and determining how they are related to one another, is of key importance to evaluating the suitability and assessing the potential difficulty that a pragmatic reuse task may entail. Developers spend much of their time while performing pragmatic reuse tasks trying to identify and to follow the structural relationships in the system; while this may sound conceptually straightforward, doing this manually is error-prone and time consuming.

2. *Understanding the source code:* While they are investigating the structural dependencies within the source code, the developer is trying to understand what role the source code under investigation has in the functionality they want to reuse. This understanding has two dimensions: (1) understanding what each structural element in the code actually does; and (2) understanding how each structural element should be managed in the reuse task (e.g., whether it should be reused or not). Ultimately, the role of the understanding task is to relate the low-level structural elements to the higher-level functional concepts relevant to the reuse task.

3. *Maintaining a mental model of the source code:* Concurrently to identifying the structural elements and their relationships within the source code and understanding what role they play in the reused functionality, the developer must remember all of the facts they have discovered and decisions they have made. Maintaining an accurate mental model of the reuse task is difficult because the source code that the developer is investigating is unfamiliar to them; trying to keep track of many new identifiers, how they are related, and what they should do with each when reused, can be overwhelming. Because of the burden this entails, the likelihood of the developer either forgetting something or remembering something incorrectly is high. These errors increase the chances of a developer investigating the same segment of code more than once or introducing inconsistencies into their mental model of the task.

4. *Testing alternative reuse hypotheses:* Naturally, the developer would like to perform their reuse task as near-optimally as possible: they would prefer that the source code they reuse represent the perfect balance between the amount of code they have reused and the amount of work they need to perform to successfully complete the reuse task. Unfortunately, the probability that the developer will make at least one poor decision while investigating unfamiliar source code is high. While they would like to know the effects that alternative decisions may have on their reuse task, the expense, in terms of manual labour and confusing their mental model, is so high that they avoid investigating alternative decisions, even if they would like to know their effects. Being able to investigate alternative reuse scenarios is essential to the developer gaining confidence in their solution and avoiding undertaking an unsuitable reuse task.

5. *Mechanical reuse operations:* Actually performing a pragmatic reuse task is a largely manual, mechanical operation. If the developer has built up an effective mental model of all the steps required to perform the task, they then have to locate each structural element they wish to reuse, copy it out of the source system, place it in their own system, and modify it as necessary to satisfy any differences between what the reused code expected of its originating system and what is provided by the developer's system. Of course, simply building up the mental model by traversing the structural dependencies within the system was a manual process in itself as the developer read the source code and manually followed the dependencies between methods, fields, classes, and files. Pragmatic reuse tasks involve a lot of low-level work that is largely

tedious for the developer, but during which it is easy to introduce errors and to forget important details.

1.2.1 How pragmatic reuse difficulties relate to task categories

The difficulties in performing pragmatic reuse tasks can be directly mapped to the five conceptual task categories. Identifying all the relevant structural dependencies without manually examining the source code in detail is error-prone. Understanding how each portion of the source code is related to the others, and what exactly the source code does, is complicated by having to deal with so many low-level structural dependencies in the process. Trying to systematically work through unfamiliar code, and maintaining a mental model of which structural elements are relevant and how they should be managed in a pragmatic reuse task, puts an inordinate cognitive burden on the developer. When working with unfamiliar source code it is hard to make the right decision the first time, every time. For this reason, a developer performing a pragmatic reuse task would like to investigate alternative reuse hypotheses in order to avoid unsuitable decisions and gain confidence in the decisions they have made; however, the amount of work required to investigate one hypothesis, let alone many, make it prohibitively expensive to investigate alternative reuse strategies. Finally, actually performing a pragmatic reuse task is a complicated mixture of the four previous issues; the developer has to manually navigate through the source code, understand what it does, remember all of the details, try to identify the best options without becoming overwhelmed, and then manually extract, copy, and integrate the relevant source code into their system. And they need to do this without forgetting details or making errors.

1.3 Enabling pragmatic reuse

All five of the conceptual categories are negatively affected by the lack of tool support for pragmatic reuse tasks and their unsystematic, ad hoc nature. To counter these two major inhibiting factors we propose a framework to make pragmatic reuse tasks more systematic, more scalable, easier to reason about, and easier to perform. We introduce the concept of a *pragmatic reuse plan* as a model for capturing a developer's intent in a pragmatic reuse task. A pragmatic reuse plan is a graph wherein the nodes are structural elements (classes, methods, and fields) and the edges are the statically derivable relationships between them (inheritance, calling, and referencing relationships). The nodes are further annotated with metadata that captures how they should be treated within a reuse task (see Chapter 5). The pragmatic reuse plan serves as a central repository for the facts and decisions relevant to a reuse task.

Pragmatic reuse tasks are typically performed piecemeal, one decision at a time, while a developer debugs the code they are reusing into existence [Rosson and Carroll, 1996]. Our framework for enabling pragmatic reuse tasks makes planning an explicit, top-level task. We hypothesize that, by explicitly planning a pragmatic reuse task, a developer can spend more time thinking conceptually

about the high-level impediments to their task and less time managing low-level details. The conceptually difficult problems often arise from architectural mismatches between the source and target systems [Garlan et al., 1995]; by planning the reuse task, we hope that the developer can identify these problems earlier in the process, before they invest much time in reusing the source code.

The pragmatic reuse plan acts as an explicit codification of the developer's mental model of their reuse task; they can glance at the plan at any time to get a high level overview of all of the structural elements they have visited, and the decisions they have made about each. Using the plan, they can see what decisions they have made and what decisions remain. The plan can help developers to keep on track; ultimately, this helps them to investigate the dependencies within the system more systematically. Another major benefit of a pragmatic reuse plan is that, by having all the decisions explicitly and consistently unified, the plan can be transformed into a set of directions describing how the task should actually be performed. While the developer could manually perform these steps, most of them can be automated, relieving the developer of large amounts of tedious, error-prone manual effort.

We have created a suite of tools, called Gilligan, that help developers plan and perform pragmatic reuse tasks. Gilligan provides an abstract view of the structural dependencies within a system and assists the developer in navigating between the dependencies while recording their decisions about how each structural element should be treated when the task is performed. Gilligan can also validate the completeness of a reuse plan, identifying decisions that should be made to complete the reuse plan. Gilligan enables developers to have a more abstract representation of the source code they want to reuse at their disposal; while they can still view the source code at any time, Gilligan strives to enable the developers to step back from the lowest-level details in order to gain a broader overview of their reuse task. Finally, Gilligan automatically enacts the reuse plan; the relevant source code for the task is extracted, transformed as necessary, and integrated into the developer's system. This process of planning and enactment can be performed iteratively, if required, to allow the developer to experiment with alternative reuse plans to converge on a solution that meets their needs.

We have evaluated our pragmatic reuse framework and model by testing several hypotheses using a suite of exploratory studies and controlled experiments using experienced graduate students and industrial developers. Our findings demonstrate that Gilligan significantly improves developer effectiveness while performing pragmatic reuse tasks.

1.4 Thesis statement and contributions

The thesis of this dissertation is that by providing developers with a mechanism to create pragmatic reuse plans in a structured way, and a methodology to semi-automatically perform the pragmatic reuse task using this plan, we can enable developers to perform pragmatic reuse tasks more quickly and with greater confidence.

The major contributions of this dissertation include:

- a model for capturing developer intent while planning a pragmatic reuse task (the pragmatic reuse plan);
- the Gilligan prototype tool that provides a visual representation of a pragmatic reuse plan and a mechanism to help developers explore and triage the dependencies within the source code they are investigating for reuse;
- evidence that developers can much more accurately identify structural dependencies using Gilligan than standard tools;
- a methodology to drastically reduce the amount of work a developer must expend while performing a pragmatic reuse task by automatically transforming the reused code according to a pragmatic reuse plan;
- evidence that Gilligan greatly reduces the number of decisions the developer must make while performing the enactment of a pragmatic reuse plan;
- evidence that developers can be significantly more effective using Gilligan to both plan and enact a pragmatic reuse task; and,
- a categorization of the key cognitive aspects facing developers as they are performing a pragmatic reuse task.

1.5 Organization

This dissertation has been written to maximize the readability of the main text by deferring many of the less important details to the appendices (see Appendix A through Appendix H). References to the relevant appendix will be given in any section in the main body of the dissertation that contains further related content in the appendices.

Chapter 2 starts with a motivating example of a pragmatic reuse task. Related work is discussed in Chapter 3. An initial survey of industrial developers' attitudes to pragmatic reuse tasks is presented in Chapter 4. Chapter 5 describes the pragmatic reuse plan, and the model that underpins it.

Gilligan, our prototype tool for supporting pragmatic reuse tasks, was iteratively developed and evaluated through three prototypes. Our first prototype pragmatic reuse planning tool was based on a visual, graph-based, literal representation of the pragmatic reuse plan (Chapter 6). Our second prototype supported pragmatic reuse planning using a tree-based representation of the pragmatic reuse plan (Chapter 7). Finally, our last prototype concentrated on supporting the enactment of pragmatic reuse plans (Chapter 8).

Each of these prototypes were evaluated immediately and the results of each evaluation, and the feedback from our participants, were heavily used to design the subsequent iteration of the tool. The evaluations given for each prototype are described along with the prototype itself (Chapters 6.3, 7.3, and 8.3 respectively). Finally, a comprehensive final evaluation of our prototype's ability to support the end-to-end pragmatic reuse planning and enactment process is described in Chapter 9.

Chapter 10 includes a discussion and an overview of future work. Chapter 11 concludes the dissertation.

Chapter 2

Motivation

Consider a developer working on a Global Positioning System (GPS) visualization system called UltiGPX. UltiGPX provides a visual representation of data collected by hikers and other outdoorspeople while they are on excursions. UltiGPX provides a simple visualization of the latitude/longitude co-ordinates of a hikers' route.

Figure 2.1 shows the UltiGPX interface; small black points represent individual co-ordinate points, larger red circles are waypoints, representing track points of interest, entered by the hiker. UltiGPX only provides a two-dimensional plan view of a hiker's track; the tool does not display the hiker's changes in elevation (an "elevation profile"). One of the hikers who uses UltiGPX considers that an elevation profile would be a useful feature and files a request for enhancement (RFE) on the UltiGPX system, asking for it to be added. With this request, the reporter includes a simple screen shot showing what such a view could look like (see Figure 2.2).

After considering the request, the UltiGPX developer decides that it would be worth adding this new feature to the system; unfortunately, the developer does not know how to construct such a view. Fortunately, the developer realizes that he knows of a visualization, within another system, that seems like it does exactly what he wants. Figure 2.3 shows a screen shot of the Azureus BitTorrent client.[1] Azureus is a peer-to-peer file transfer client that happens to have a network throughput visualization that is similar to the initial feature request. The visualization provides an attractive graph view that uses gradients, has a trend line, and provides additional information on its axes. The UltiGPX developer decides that the Azureus view provides much of the functionality he would like to have within his program and decides to investigate reusing this view instead of writing one himself from scratch.

The UltiGPX developer realizes that while the network throughput graph is visually similar to what he requires, he does not know if it will be technically feasible to perform the reuse task, although he envisions that the final result could look something like the mock-up shown in Figure 2.4. A significant hurdle haunts the developer: Azureus was designed to support peer-to-peer file down-

[1] http://Azureus.sf.net v2.4.0.2

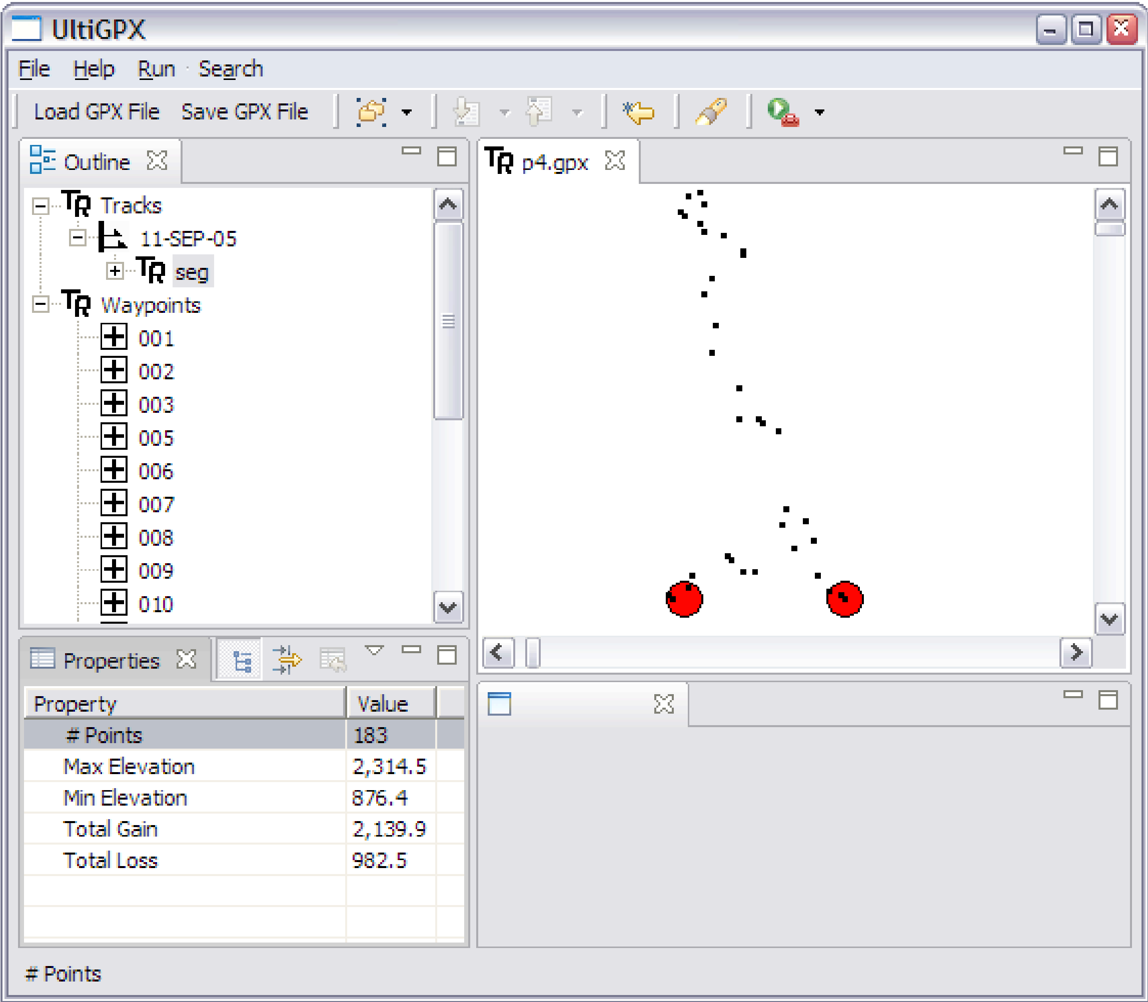


Figure 2.1: The UltiGPX application. The developer has left space for an alternative data view in the lower right corner.

loading, not to visualize GPS track points or to provide reusable APIs for its user interface widgets. Visually, the Azureus feature looks exactly like how the developer would like the feature to look; however, it seems unlikely that a feature providing real-time network visualization would be appropriate to use within a static GPS-data viewing application.

The developer wants evidence that he could successfully reuse the functionality from Azureus; he does not want to rely on high-level intuition. Ultimately, he wants to understand how dependent the graph visualization feature is on the rest of the Azureus system. If the source code he wants to reuse is tightly coupled with those aspects of Azureus that are irrelevant for his task, reusing the code may be more difficult than writing the feature from scratch or finding another alternative to reuse. In order to do determine how coupled the code is, he investigates the source code manually within an integrated development environment (IDE). First, he uses the IDE search features to locate a part of Azureus

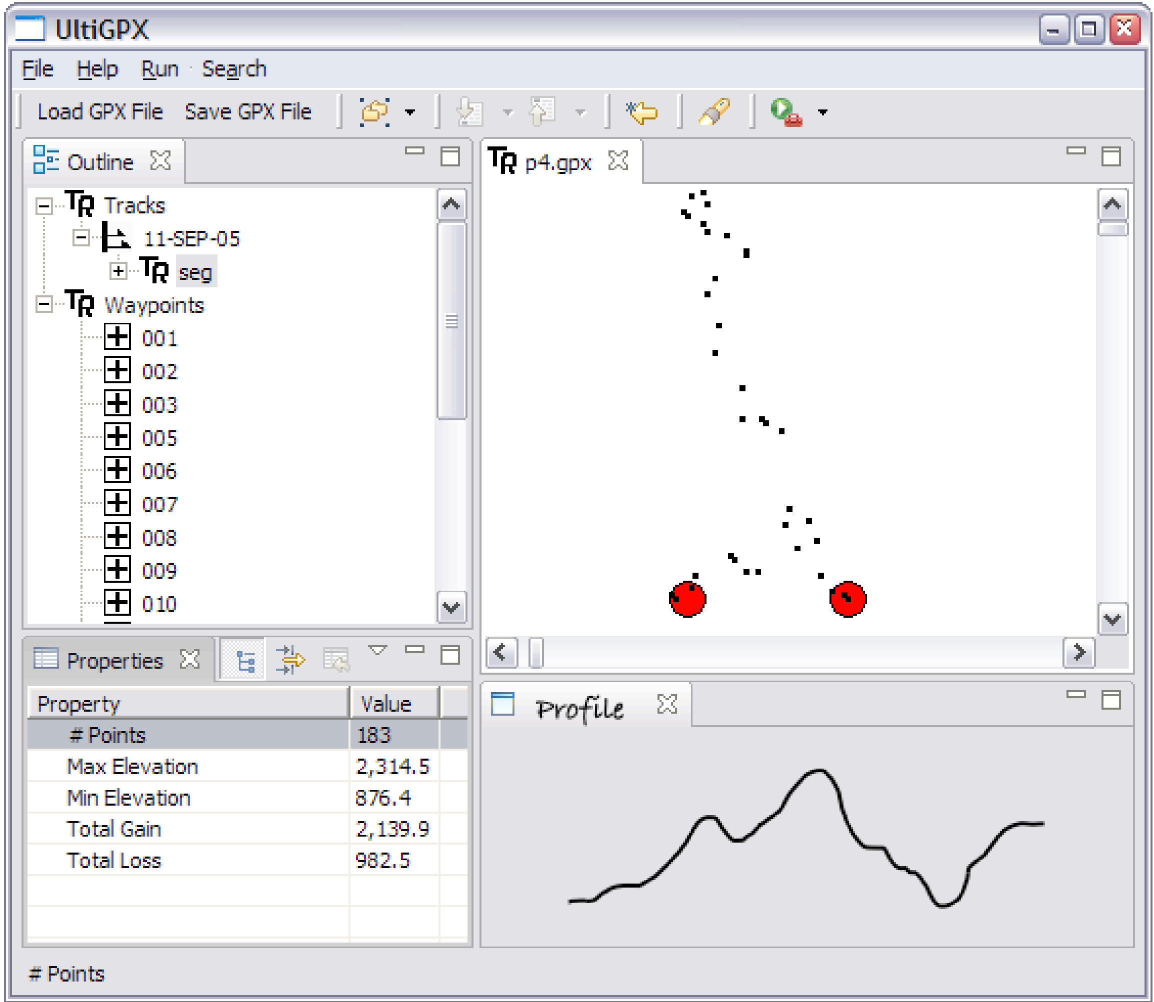


Figure 2.2: The user who filed the request for enhancement provided this mock-up of the type of view they would like to see UltiGPX provide.

involved in network visualization; this search quickly leads him to the `org.gudy.azureus2.ui.swt.components.graphics` package, in which `SpeedGraphic` seems likely to be the most relevant class. The developer starts by scrolling through the 322-line class, trying to identify which portions of the class are relevant to his reuse task.

The `drawChart(..)` method sounds to the developer like the most relevant method within `SpeedGraphic` so he begins his exploration task there. To investigate the implications of each dependency in this 82-line method, the developer must examine each statement to determine which types are used, fields are referenced, and methods are called. He then needs to look at each type to determine its dependencies and to decide whether or not to reuse those types in addition to `SpeedGraphic`. In the `drawChart(..)` method, 14 different types are referenced. After navigating through 14 different files corresponding to these types, he determines that 7 of the types are common

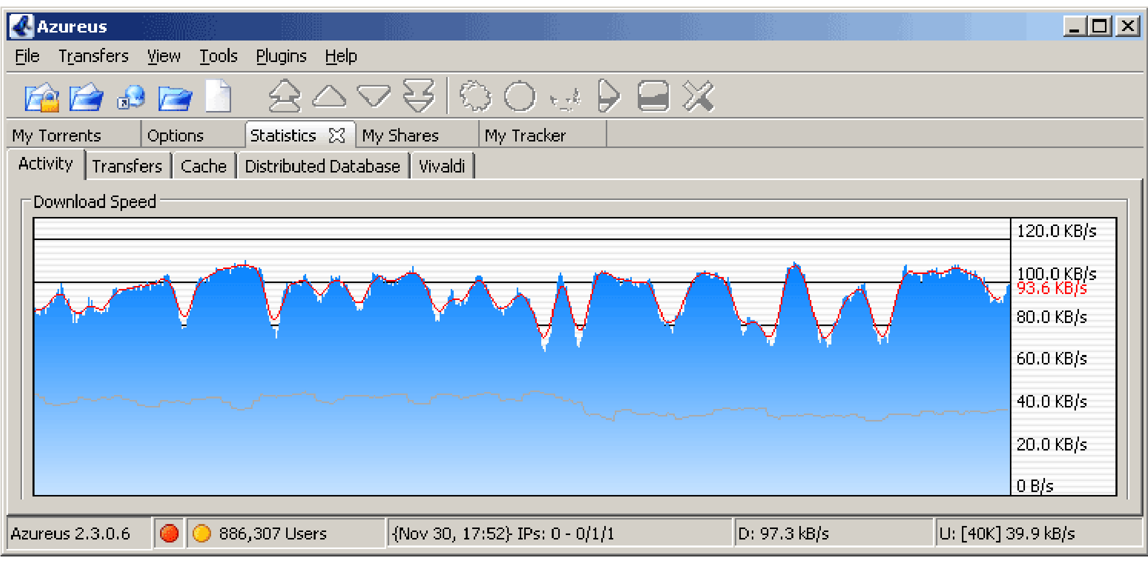


Figure 2.3: The network throughput view in the Azureus BitTorrent program. The developer remembers this view as being visually similar to what he would like to provide including axes, gradients, and a smoothing line.

to both UltiGPX and Azureus (they both use the SWT framework) which means that these dependencies are already satisfied within UltiGPX and do not need to be reused; however, the developer must look more critically at the 7 remaining types to determine their relevance to his task.

`SpeedGraphic.drawChart(..)` makes one call to both the `enter()` method and the `exit()` method from the `AEMonitor` class. Thinking that these calls are related to his task, the developer decides that he should reuse the methods but decides to investigate `AEMonitor` further, just to be sure. The `AEMonitor` class seems simple enough but when the developer looks at the class it extends (`AEMonSem`) the developer finds more than 700 lines of threading code that does not seem very relevant to his task. Instead of reusing the `AEMonitor` class, the developer decides that he will remove the method calls to try to avoid reusing dependencies that are heavily coupled with the core of Azureus. Similar situations arise for `COConfigurationManager` and `ParameterListener` which are involved with the Azureus preferences architecture (and like `AEMonitor` are tightly integrated with scores of other classes within Azureus). The developer does decide to maintain the dependencies to the `Scale` class as well as `SpeedGraphic`'s supertypes, `ScaledGraphic` and `BackgroundGraphic`.

As the developer is investigating the code, he notices many references to constants in the `Colors` class. These references roughly duplicate functionality he already has in UltiGPX: he defines his own colour constants. Instead of reusing the 28 fields in the `Colors` class, he remaps the 13 references to the 9 colour constants the code he wants to reuse references to his own colour constant fields within UltiGPX.

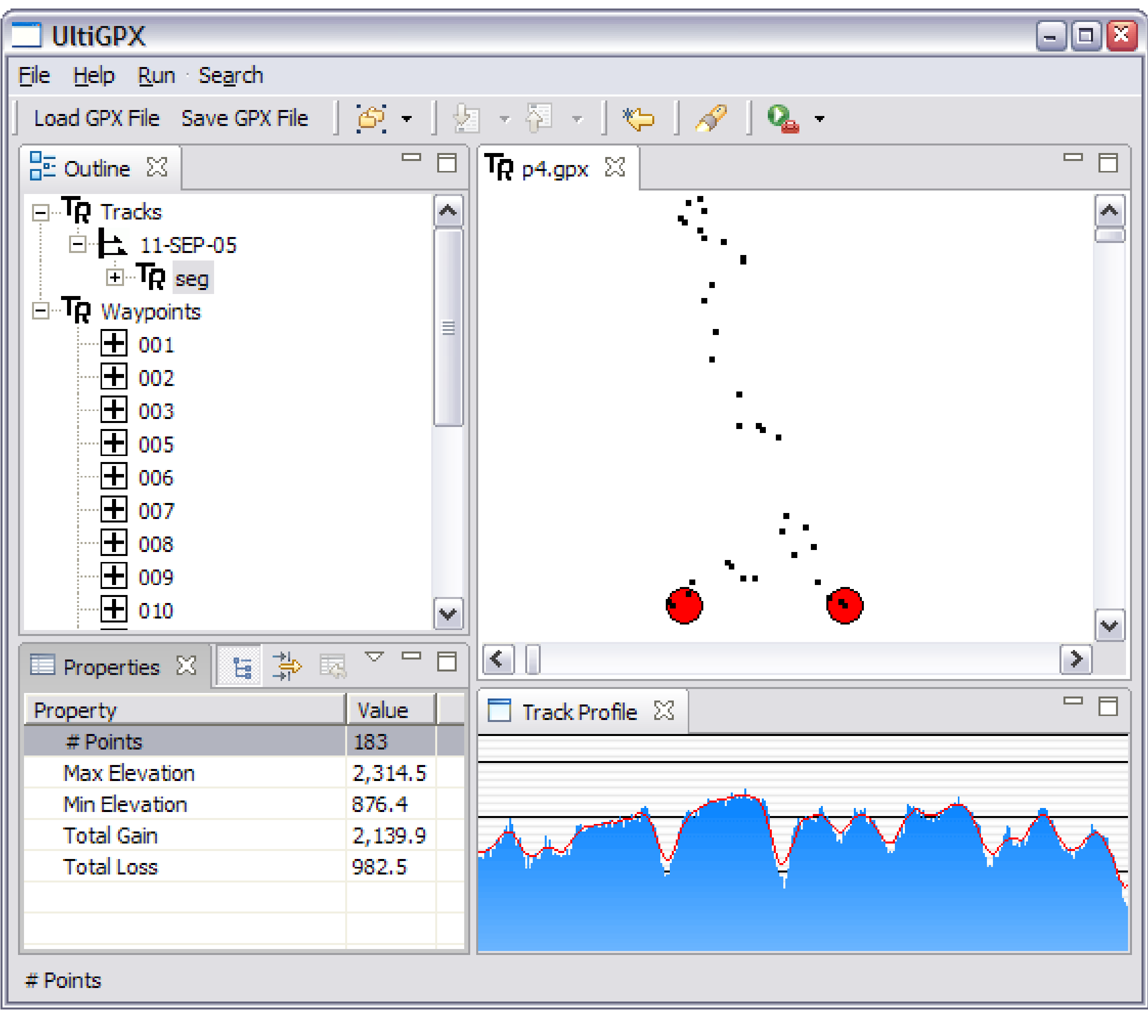


Figure 2.4: A mock-up simulating how UltiGPX could look if the Azureus network throughput view reuse task was successful.

Figure 2.5 provides a simplified class diagram corresponding to the developer's mental model of the reuse task (the final size of the task is given in Table 2.1). Green structural elements represent those elements that the developer wants to reuse. Red elements represent those elements that the developer does not want to reuse. Blue elements represent those elements that are analogous between UltiGPX and Azureus (namely the colour constants for this task) that the developer would like to remap. The yellow nodes represent common functionality between the two systems (due to shared libraries, in this case). The developer will have to make modifications to the source code to remove any red nodes, and on any edge between a green node and red or blue nodes; the class diagram does not capture the extent of work this could entail: if the developer had rejected a commonly-called method he could potentially have to make hundreds of edits throughout the system.

While the class diagram representing the reuse task is relatively simple, the developer had to

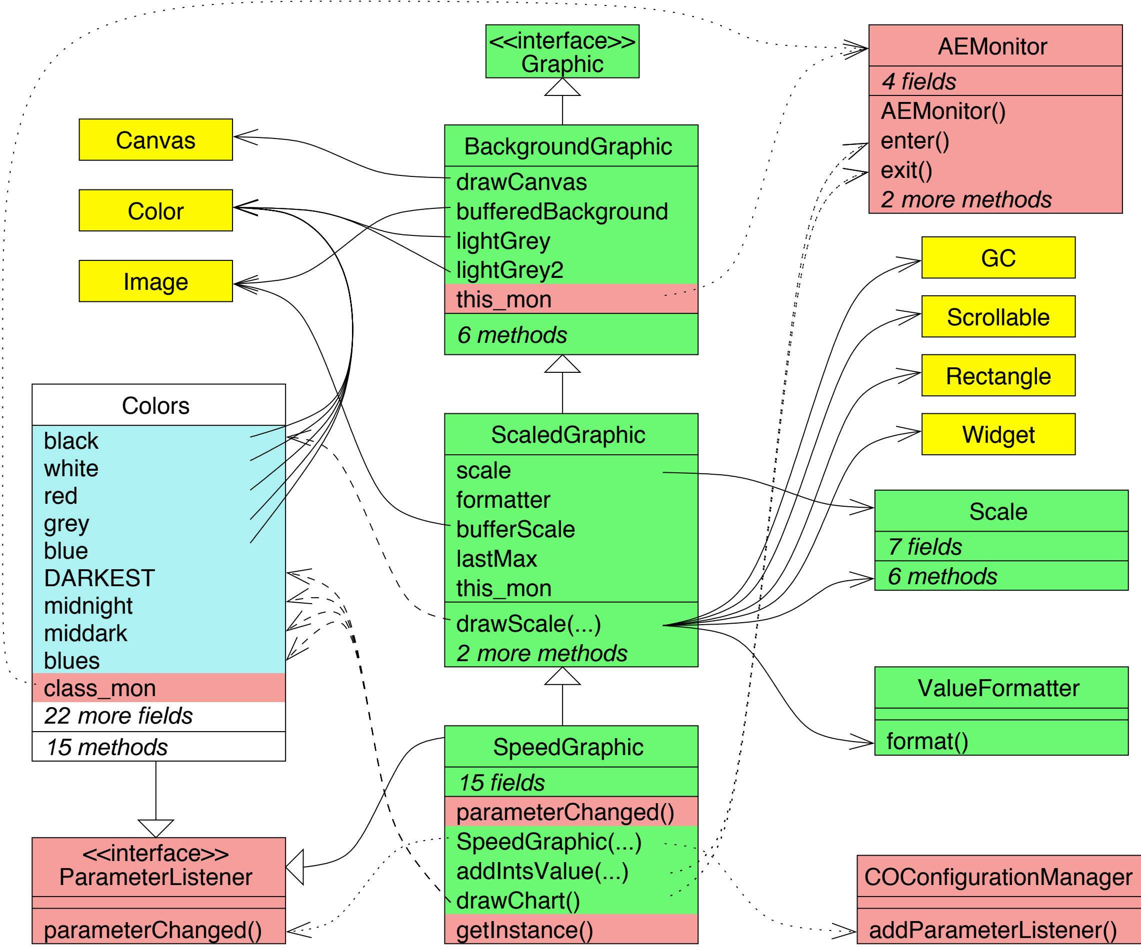


Figure 2.5: Simplified class diagram for the Azureus reuse task. The green represents structural elements the developer will reuse, yellow represents common elements, blue represents elements that are similar to those the developer already has in there system, while red represents functionality the developer does not want to reuse.

Element	Count
Classes	6
Fields	30
Methods	25
Lines of Code (LOC)	373

Table 2.1: Scope of the Azureus network throughput graph pragmatic reuse task after the extraneous functionality has been removed.

investigate more than 20 different source files to make decisions about the importance of each of them. Tracking these decisions while negotiating unfamiliar source code can be daunting. While he may not be able to enumerate every decision he has made as the class diagram suggests, he has a sense

that the task should be manageable and begins to carry it out. To actually perform the task he has to copy those relevant classes from Azureus to UltiGPX, modify the copied files as necessary to remove rejected elements and calls and references to these rejected elements, and to update any remapped elements to elements within UlitGPX. While carrying out the task is conceptually straightforward, it is difficult for the developer to remember all of the decisions he made while navigating between these various source files. Indeed, when he started to do the task, he had to revisit several files to remember what decisions he had made. One problem that the developer had while investigating the code is that he never actually knew if he was "done" his investigation; he may have missed an important dependency when he was navigating the various files and may not have found out about it until he actually attempted the reuse task. The main problems with this task are that it would be easy to dismiss it out of hand due to the domain differences between Azureus and UltiGPX, and the fact that it required a lot of manual effort on the developer's part; this effort was an act of faith on his part as he did not know if the task would successful until he had completed it.

The `SpeedGraphic` functionality has been widely tested and deployed and can be considered mature. It has existed for more than 4 years and has undergone more than 40 revisions. The code has also been widely deployed as Azureus has been downloaded more than 200 million times. While the task only ended up reusing 373 lines of code (LOC)[2], reusing the code took significantly less time than writing it from scratch; additionally, the fact that the code was of high quality made the developer feel more comfortable than he would with his own first attempt at such functionality. Figure 2.6 provides a final screen shot of UltiGPX with the profile feature added by reusing the Azureus code. The gradients, smoothing line, and axes all combine to provide a visually-appealing and functional addition to the UltiGPX system.

Ultimately, this reuse task was successful; the developer was able to add the elevation profile feature to UltiGPX without having to write the code from scratch. At the same time, he did not reuse the code in a black box manner, the code was reused pragmatically by copying it from Azureus, pasting it within UltiGPX and modifying it to suit its new context. Refactoring the original code was not an option here: the UltiGPX developer is not also a member of the Azureus project. Even if he were, it would be unlikely that a patch to Azureus to provide a reusable version of the `SpeedGraphic` functionality would be accepted as the goal of Azureus is to be a BitTorrent client, not to be the source of reusable graphic widgets. By reusing the source code from Azureus the developer was able to add new functionality that leveraged code that had been widely deployed for several years; this saved him time and ensured that the new feature was of high quality.

[2]All of the LOC counts presented in this dissertation are computed by counting non-comment source lines (NCSL); this was performed by the Eclipse Metrics Plug-in v. 1.3.6 according to the standards outlined by Henderson-Sellers [1996].

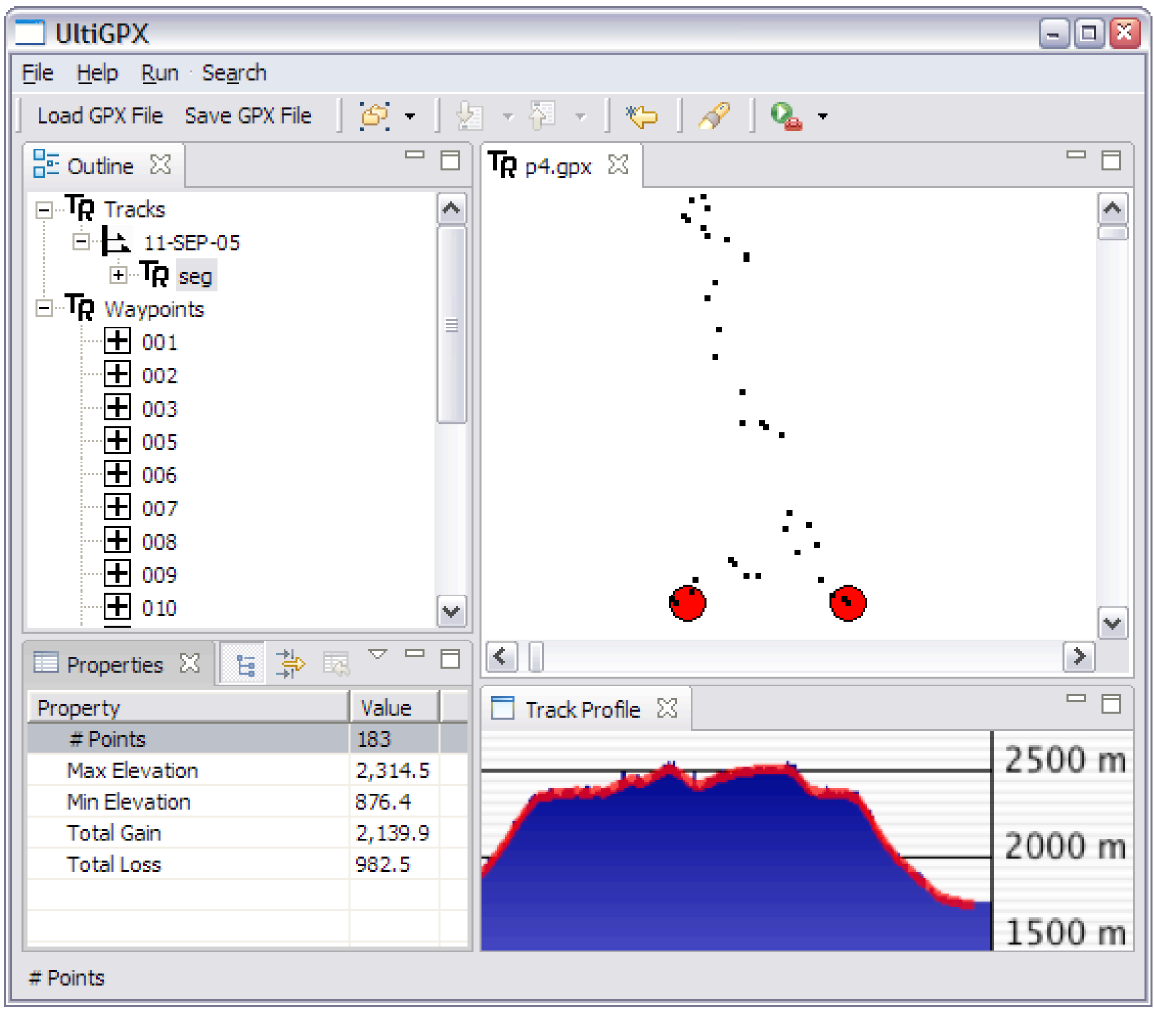


Figure 2.6: The final result of reusing the Azureus network throughput view within Azureus. The developer has chosen to remap several of the colours to better match those already in UltiGPX.

2.1 Alternative to pragmatic reuse

The main alternative approach for completing this task without pragmatically reusing an existing feature would be to write the source code from scratch. As the reused code is of reasonably high quality, the LOC count (373) does not seem excessive. While measuring productivity in terms of LOC per unit of time has been generally disregarded as an effective measure (e.g., [Lawrence, 1985; Berg et al., 1995]) we use it illustratively here.

If you consider that the 373 LOC in the existing solution contained on average 4 words, a developer typing 45 words per minute would take over 45 minutes to write this code from scratch. This of course ignores the fact that developers cannot be expected to write code as fast as they can type and that the likelihood of there being defects with code written at that speed is high.

Alternatively, we can consider that developers have been recorded as averaging between 8 and 50 LOC per day (e.g., [Drake, 1996; Prechelt, 2000; McConnell, 2004]). At this rate, assuming an 8-hour work day, this feature would take between 7 and 45 days to complete. In my personal estimation, I believe this feature would take at least two days: one full day to write it initially and one full day to debug and refine it. For small, well-understood features, a developer may be able to quickly write the code from scratch, but for larger features the risks and costs associated with writing source code from scratch quickly grow. The intent of this simple estimate is not to show that pragmatic reuse is always faster than writing source code from scratch, but to remain aware of the high costs associated with creating new software.

In either case, this functionality can be completed pragmatically in less than 45 minutes by reusing code that is of high quality and has been used by millions of people; thus, pragmatic reuse is apparently *more productive* than creating the code from scratch, although empirical evidence is still needed.

2.2 Summary

In this reuse task, a developer wanted to add a specific piece of functionality to his system. He knew of an existing system, from another domain, that provided functionality similar to what he wanted, so he decided to investigate the source code associated with this feature as a potential reuse candidate. His primary investigative goal was to identify how coupled the feature he wanted to reuse was on the rest of its system; to perform this investigation he manually navigated through many files in the system, tracing the structural dependencies from file to file to build up a mental model of his task.

After a highly-manual investigation he decided to proceed with the task, copying the code associated with the feature into his own project, then modifying it to work within its new context. This copy-and-modify process was time consuming but resulted in the functionality being successfully added to the developer's system. Even though the reuse task may have seemed ill-advised initially due to the domain mismatch of the systems, and investigating the task and performing it required a lot of manual and mental effort on his part, the reuse task ended up taking much less time than writing the feature from scratch.

Pragmatic reuse provides developers with a mechanism that enables them to leverage existing high-quality code from existing systems within their own projects. These tasks can be completed significantly faster compared to creating the functionality from scratch; by most measures, reusing validated, deployed, source code is preferable to writing new source code.

Chapter 3

Related Work

Software reuse has a well-established history in both the research literature and industrial practice [for example, Mcilroy, 1968; Poulin et al., 1993]. Due to the broad scope of software reuse, relevant research is found in many different fields including software engineering, programming languages, information retrieval, information visualization, and many fields that utilize domain-specific reuse mechanisms. The diversity of these fields reflects the variety of issues surrounding the three primary phases of a software reuse task (locating an artifact for reuse, specializing it for the reuse context, and performing the reuse task).

Research efforts relevant to this dissertation have been categorized into five primary categories: software reuse (Section 3.1), source code transformations (Section 3.2), program understanding (Section 3.3), software visualization (Section 3.4), and cognitive aspects pertinent to reuse (Section 3.5). This chapter outlines relevant related work and differentiates the research in this dissertation from previous research efforts.

3.1 Software reuse

Software can be reused in either a black-box or white-box manner.[1] This high-level categorization differentiates approaches that enable the developer to modify the internal operation of a reusable artifact (white-box reuse) and those that do not permit the internals of the artifact to be changed (black-box reuse).

At a high-level, each of these approaches have general strengths and weaknesses. Black-box reuse enables developers to reuse an artifact without requiring the developer to consider how the artifact works; the main downsides of this approach are that black-box reuse tends to be less flexible as the developer cannot modify the artifact to suit their specific needs. In contrast, white-box reuse enables the developer to modify an artifact to suit their specific needs at the expense of the additional effort required to understand and modify the reused artifact.

[1] A third category, grey-box reuse, is sometimes mentioned; in this dissertation it is considered a form of white-box reuse.

3.1.1 Black-box reuse

The majority of reuse literature emphasizes designing software in a reusable fashion, for example, in object-oriented programming [Johnson and Foote, 1988], frame-based reuse [Bassett, 1997], domain-specific language-based approaches [Neighbors, 1989], component-based approaches [Mcilroy, 1968; Meyer, 1999; Szyperski, 2002; Mezini and Ostermann, 2002; Ravichandran and Rothenberger, 2003; Estublier and Vega, 2005], and in software families and software product lines [Dijkstra, 1972; Parnas, 1976; Krueger, 2000].

Black-box reuse approaches rely on the systematic application of a software reuse policy within an organization. This is due to the large cost associated with these approaches and the potentially large payoffs that can be gained by applying them. To enact a black-box reuse program, an organization requires a large library of reusable software assets and often must also make an investment in adding their own assets to the repository as they create them. Much of the effort in black-box reuse is a result of populating, maintaining, and searching the library of reusable artifacts. The scope of the reused asset tends to be bimodal for black-box reuse. The reuse usually involves small assets, such as individual functions [Kim et al., 2005], or very large assets such as whole components [Weide et al., 1991].

Other systems have looked at creating reusable components from existing code. CARE [Caldiera and Basili, 1991] takes a metrics-based approach to identifying reusable components. Component rank attempts to locate the most relevant components to reuse for a particular query using a weighted-graph model approach that considers structural edges and component usage patterns [Inoue et al., 2003]. Lanubile and Vissagio [1997] have also investigated extracting reusable code from existing systems using a program slicing approach, although it is unclear if this approach has been implemented.

CodeGenie [Lemos et al., 2007] extracts code slices from existing systems based on test cases selected by the developer. While the developer cannot influence specific aspects of the slice, they can consider the output of several different slices to determine which best provides the functionality they would like to reuse.

In contrast, pragmatic reuse tasks are more opportunistic; they arise when an organization has not made the investment in a systematic reuse policy—or even if they have [Selby, 2005]. A developer simply decides that they want to reuse some existing functionality, regardless of whether it has been designed in a reusable fashion or not, and performs the reuse task manually. Pragmatic reuse tasks do not require a library of reusable assets but are highly dependent on developer experience and their social networks to identify situations where similar code may exist. Pragmatic reuse tasks can be of any size, but are typically limited by the developer's ability to fully understand their reuse task; as such, they tend not to be as large as the largest black-box tasks, and are particularly suited for medium-scale reuse tasks.

3.1.2 White-box reuse

Selby [2005] analyzed 25 projects at NASA and discovered that 32% of the modules within those projects were reused from prior projects. Of these reused modules, 47% required modification from their original form, while 53% required no modification. This study showed that a significant proportion of reuse instances involve source code modification; this further motivates supporting these types of reuse tasks.

Parsons and Saunders [2004] determined that developers were able to perform tasks by anchoring their understanding to existing code and adjusting the code to meet their needs. While this evaluation was only tested for one small case (albeit with many developers), it is an encouraging endorsement for the white-box reuse paradigm.

Several approaches have investigated small-scale white-box reuse. Toomim et al. [2004] argue that there are cognitive costs associated with abstraction and that copy and paste development provides a mechanism to avoid some of these costs. Their linked editing approach allows developers to copy artifacts while enabling the copied versions to be synchronously edited by changing only a single instance of the copied code. The CReN tool [Jablonski and Hou, 2007] provides tracking and identifier renaming support when code is reused through copy and paste; the primary intent of CReN is to alleviate defects caused by inconsistent identifier names. These approaches are both designed to address the issue of copying code *within* a single system; the intent of pragmatic reuse tasks is not to clone small-scale functionality within a system but to enable developers to reuse larger-scale functionality from external systems more effectively.

Cottrell et al. have developed the Jigsaw tool that allows one method to be copied and integrated into another existing method [Cottrell et al., 2008a; Cottrell, 2008; Cottrell et al., 2008b]. Their approach effectively integrates methods, properly identifying the commonalities between a source and target method and ordering the elements within the target method. Jigsaw provides more comprehensive support for the integration of one method into another than Gilligan does; however, Jigsaw is heavily reliant on contextual information provided by the developer regarding where the reused code should be placed. In addition, Jigsaw operates on a smaller-scale than our approach as it only integrates individual methods; it does not integrate whole classes or features.

The Strathcona Example Recommendation System was developed as a means to locate structurally-relevant source code examples for a developer based on some fragment of source code they had selected [Holmes and Murphy, 2005; Holmes et al., 2006]. The intent of the system was for a developer to locate an example that provided the functionality they wanted from which they could then copy the source code into their own system. Other example recommendation systems such as ParseWEB [Thummalapenta and Xie, 2007] have similar intent to Strathcona. These systems are different from the work presented in this dissertation as they only locate examples; they do not help the developer investigate or reuse the source code.

3.1.3 Code clones

Pragmatic reuse can be perceived as a means to easily create clones within a system; while this is true, the intent of pragmatic reuse is to reuse functionality from external systems rather than introduce clones throughout a single system. Kapser and Godfrey have created a taxonomy [2006] of the different kinds of cloning encountered in their study of several large software systems. One type of cloning they identify as "replicate and specialize"; is essentially white-box reuse.

While clones have been perceived negatively in the past, research has found that they are frequently short-lived, and when they are long-lived they are not easily refactored [Kim et al., 2005]. Short-lived clones are those that are reused and then modified to meet the new system's requirements. Non-refactorable clones indicate that the original API could not be refactored to meet the requirements of both the old and new usage. These cases are no worse than implementing the features from scratch but the developers still get the added benefit of having reused code. One problem with reusing code in this manner, however, is that when bugs are fixed in the original source they are not automatically propagated to the reused versions; support for this type of process is an active research topic [Kim and Notkin, 2006; Duala-Ekoko and Robillard, 2007].

Cordy [2003] provides many reasons by which industrial organizations rationalize reusing code via clones instead of refactoring. Two primary rationales are raised. First, refactoring does not directly contribute to an organization's financial situation; while refactoring source code may have future benefits, these are not realized immediately and can be difficult to make a sound business case for. Secondly, the risk associated with refactoring a system may not be acceptable. This risk is manifested in both the changes to the system itself, but also by exposing multiple parts of a system to a single module, instead of each part having its own that it can change independently, isolating subsystems from one another as much as possible. Cordy argues that the decision to reuse source code via clones is well rationalized by the developers who perform those tasks and that they use this technique to ensure specific properties hold for their systems.

3.2 Program understanding

An essential aspect of performing a pragmatic reuse task is locating and understanding the source code that is to be reused. Feature location approaches help developers locate these portions of the source code.

These approaches can use static information gleaned from the source code, dynamic execution information, or a hybrid combination of the two. To reduce the overhead required to plan a pragmatic reuse task, only static approaches will be considered here. Ciao [Chen et al., 1995] and Rigi [Müller and Klashinsky, 1988] both statically analyze the source code of a system and allow a developer to navigate the structural dependencies contained therein. Chen [2000] investigated feature location using a graph-based technique, although this approach was purely a generic program understanding tool.

Many approaches have appeared in recent years that automatically or semi-automatically identify the extent of features in source code (e.g., [Eisenbarth et al., 2003]). Such approaches could be used as a starting point in a pragmatic reuse task; however, given that features of interest are often not well-encapsulated, an intricate and inexact decision-making process is still needed to draw the boundary between the feature and the rest of the system. Previous work on feature location does not aid in that task.

The FEAT [Robillard and Murphy, 2002] tool helps developers create descriptions of scattered software, called concerns. A concern in FEAT is a graph where the nodes are software elements and the edges are the relationships between them. A pragmatic reuse plan is also a graph, but extends the nodes to include triaging information that capture how the structural element should be managed when the pragmatic reuse task is performed. Recent work by Robillard [2005] has investigated providing tool support that can recommend relevant program elements to the developer based on the elements they have previously investigated. This technique could be directly applicable to the planning aspect of pragmatic reuse tasks.

One major difference between Gilligan and each of these program understanding approaches is that Gilligan was not developed as a generic program understanding tool; each design decision considered while constructing Gilligan was made to enhance the process of creating a pragmatic reuse plan. This task specificity is an essential component to Gilligan's effectiveness.

3.3 Program transformation

Transformation-based approaches to reuse were prevalent in the 1980s, for example, that of Feather [1989]. Such approaches were based on the notion of formally correct refinement, thus requiring compilable programs and (usually) formal specifications; neither is available in the context of pragmatic reuse. Likewise, metaprogramming [Kiczales et al., 1991] and most generative programming techniques, such as aspect-orientation [Lieberherr et al., 1994; Kiczales et al., 2001], require complete and compilable source code prior to transformation. Lower-level transformation approaches (lexically- or syntactically-based), such as DMS [Baxter, 2002] or TXL [Cordy, 2006], could be used to avoid the difficulties in dealing with uncompilable code; however, a tool like Gilligan would still need to augment them to perform the semantically-aware operations needed for the enactment of reuse plans.

Various approaches attempt to adapt code for use in a novel context. The Adapter object-oriented design pattern [Gamma et al., 1994] adapts classes or objects to conform to a required interface, but maintains all the dependencies of the original classes or objects; to enact a pragmatic reuse plan, dependencies sometimes need to be eliminated or replaced. Approaches like that of Gouda and Herman [1991] and of Yellin and Strom [1997] automatically adapt components to new contexts; however, they require complete, formal specifications to operate that are not applicable to the lightweight process inherent in pragmatic reuse tasks.

Automated support for program restructuring [Griswold and Notkin, 1993] (also known as refactoring [Opdyke, 1992; Henkel and Diwan, 2005]) is either limited to meaning-preserving transformations that require compilable source code to operate correctly, or simple operations like replacing the name in a declaration and its corresponding references within a method body (the Eclipse IDE[2] provides such support).

While many approaches have advocated refactoring code into reusable application programming interfaces (APIs), this is not always possible. The original code may no longer be maintained or its maintainers may not be willing to refactor the code to meet the new requirements. Indeed it has been shown that reused code must be frequently modified in some way to work within its new context [Selby, 2005]. Frakes and Kang [2005] note that dedicated reuse strategies within companies require a large up-front cost that must be justified in terms of business goals. They also found that most software systems are variants on pre-existing systems. As new systems are extensions of the old, it is natural that pragmatic reuse will take place in situations where the new requirements do not align perfectly with the old.

3.4 Software visualization

A variety of program comprehension approaches are based on graph visualization (e.g., [Müller and Klashinsky, 1988]). In attempting to be as general-purpose as possible, they tend to be ill-suited for specific tasks [Schafer et al., 2006]. They also tend to ignore the particular needs of industrial developers [Reiss, 2005] in investigating pragmatic reuse tasks and quickly deciding whether or how to pursue them. We impose a particular process model and its related navigational strategy on developers using our approach; despite the fact that this would not be a good design choice for a general-purpose program understanding tool [Storey et al., 2000], we believe that the narrowly targeted application eliminates the need to permit a range of navigational strategies.

Several visual techniques exist to view graph structures as trees, e.g., TreePlus [Lee et al., 2006]. TreePlus advocates the *Plant a seed, watch it grow* metaphor for graph visualization, starting from a point and working outwards—essentially, propagational navigation. It is unclear how the learning burden required to learn how to interact with this visualization approach would affect developers performing a pragmatic reuse task. While initial prototypes of Gilligan used a visual graph-based metaphor, later versions changed to representations using standard tree user interface (UI) widgets in order to reduce the overhead required to learn how to use the tool.

3.5 Cognitive aspects of reuse

Many of the problems impeding the successful reuse of software are cognitive [Fischer, 1987]. In order to overcome these cognitive impediments, Fischer argues that more information is not needed,

[2] http://eclipse.org

but that the current information must be structured more effectively. He further argues that tools must support the developer in safely investigating alternative reuse scenarios. In terms of pragmatic reuse tasks, these observations are especially pertinent: manually experimenting with alternative reuse solutions is prohibitively expensive; we believe that adequate tool support can help reduce this burden and encourage developers to make better decisions by understanding the differences between different alternatives.

Parsons and Saunders [2004] determined that developers were able to perform tasks by anchoring their understanding to existing code and adjusting the code to meet their needs. While this evaluation was only tested for one small case (albeit with many developers), it is an encouraging endorsement for white box reuse approaches. By helping developers create a concrete reuse plan, we want developers to anchor their reuse activity within the existing system so they can better understand how the code needs to be adjusted without being overwhelmed by all of the low-level details the task requires.

de Alwis and Murphy [2006] cite three key shortcomings of current IDEs that can contribute to disorientation amongst expert developers: (1) the lack of connecting context when switching between files; (2) the inability of the developer to see all of the information pertinent to their task at once; and (3) the absence of support for diverging from a main task to pursue sub-tasks. We believe that these three shortcomings are strongly related to the central focus of development activities on the source code editor. Sen [1997] also notes that determining the order of the steps in a reuse task a priori is not possible; this further supports de Alwis' findings for the support of sub-tasks. Our approaches will leverage abstractions of the source code to reduce the disorientation caused by these three problems.

Two issues that can impede a developer's understanding of a tool are its internal complexity, that is, the difficulty to understand what the tool is doing to determine how to best utilize it, and its deployability [Sim and Storey, 2000]. To address these issues, we aim to tailor our tools to supporting only pragmatic reuse tasks: by supporting only one specific type of task we hope to make it simpler and easier to understand. We will address interoperability of our tool with existing environments and deployability by integrating our tools into the Eclipse IDE as much as possible.

3.6 Summary

The related work demonstrates that industrial developers perform pragmatic reuse tasks. Developers perform these tasks both opportunistically, and as an explicit mechanism to provide specific benefits for their systems. Pragmatic reuse tasks are performed in an unsystematic, ad hoc fashion; this makes them error-prone and limits how well these tasks scale to larger pieces of functionality. Software reuse tasks are complex enough without the developer having to additionally reason about and understand the unfamiliar source code they are trying to delineate from its system and reuse. The literature provides scant evidence pertaining to how developers actually perform pragmatic reuse tasks.

Chapter 4

Industrial Applicability Survey

Before investigating the concept of pragmatic reuse further, we wished to verify that industrial developers were reusing code in practice. To do so, we surveyed industrial developers from several different organizations. The primary research question the survey sought to answer was, "Do developers actually perform pragmatic reuse tasks?" The secondary research goal was to gain a greater understanding about how they thought about and performed these tasks and to identify any barriers they experienced when performing these tasks.

We surveyed 12 developers from 6 different companies, each of which was in a different industrial domain. Each respondent was employed full-time in an active development role; their experience ranged between two and twelve years of industrial software development (with an average of over 6.5 years). This survey investigated how developers think about, and perform, pragmatic reuse tasks. The survey consisted of two parts: (1) a set of 32 questions that the developers answered using a seven-segment Likert scale; and (2) a set of 10 open-ended questions to which developers were encouraged to freely state their opinions about software reuse in general, and pragmatic reuse in particular.

The Likert scale questions can be found in Table 4.1; the complete text of the questionnaire is included in Section A.1 (p. 135). All the responses to the Likert-scale questions can be found in Table A.1 with a summary of the responses given in Table A.2. The results of this survey have been previously published [Holmes and Walker, 2007a].

ID	Question
Development	
D1	I primarily develop new features.
D2	I primarily maintain existing features.
D3	I have flexibility in choosing how to complete my development tasks.
D4	Creating reusable software is encouraged in my organization.
D5	Reusing software is encouraged in my organization.
D6	My organization has a large amount of code available to be reused.
D7	Portions of features I am developing already exist in other software systems (and I have access to the source code).
Environment	
E1	I use an IDE (while working with code).
E2	I use advanced IDE tools (e.g., refactoring support, type hierarchy navigation).
E3	My primary development environment is Eclipse.
E4	My primary development language is Java.
E5	I use program understanding tools (beyond those provided by Eclipse itself) to help me complete my tasks.
E6	I prefer using a suite of general purpose tools that I can apply to many tasks.
E7	I prefer targeted tools that are designed to help me accomplish specific tasks.

Reuse	
R1	I have reused source code (any scope).
R2	I have reused whole classes.
R3	I have reused whole features.
R4	The reuse process as outlined on page one is similar to how I think about the reuse process.
R5	When I reuse source code I am careful to only reuse exactly those portions that are relevant to the task I am completing.
R6	When reusing source code I worry about not fully understanding the code I am reusing.
R7	I rely on IDE tools to help me complete reuse tasks.
R8	I reuse source code to save myself time.
R9	I reuse source code to increase the reliability of my code.
R10	I reuse source code to increase the robustness of my code.
R11	Given the choice of implementing a feature or reusing one from an existing system, I would choose to roll my own (aka. not reuse).
R12	The size of the feature I am working on would influence my decision to try to reuse it or to re-implement it from scratch.
R13	Keeping track of the relevant details of a piece of source code while navigating its text can be difficult.
R14	Understanding what dependencies a feature has on its context is important for me to determine whether I should reuse it.
R15	I am more inclined to reuse smaller features about which I can have a complete understanding, than larger features that are harder to reason about.
R16	I prefer reusing smaller features because it is difficult to both build an understanding of a complex reuse task, and carry it out.
R17	Reusing smaller features provides less benefit to me than reusing large features.
R18	The definition of reuse outlined on page one is reasonable to me.

Table 4.1: Likert-scale questions from industrial questionnaire.

4.1 Survey Results

Upon analyzing the survey results, four main themes were identified. These themes mainly arose from the Likert-scale questions; the developers' responses to the open-ended questions were used to support them. Some selected statements and responses from the Likert scale questions are summa-

rized in Table 4.2, while the histograms corresponding to them have also been included (p. 30).

1. *Developers perform pragmatic reuse tasks.* Developers agreed that they had reused source code (Table 4.2, R1) and that these reuse tasks frequently encompassed whole classes (Table 4.2, R2). In the long answer section the developers indicated that their reuse tasks usually ranged from 4 lines to 50 (several methods or a portion of a class), but sometimes included whole classes (up to 1000 lines). Their comments also indicated that reuse of this nature frequently occurred while prototyping new features, or in the early stages of a project when functionality was incorporated from existing products. Developers generally agreed they would rather reuse a feature than re-implement it themselves (Table 4.2, R11).

2. *Developers have access to large amounts of code.* Our respondents strongly agreed that their organizations had large repositories of source code available to them to reuse (Table 4.2, D6). Additionally, they reported that portions of the features they developed were available in other systems for which they had access to the source code (Table 4.2, D7). Several respondents listed porting activities (making source code that works on one platform function properly on another) as a frequent rationale for performing pragmatic reuse tasks; for these tasks the respondents generally had access to the source code for one platform and had to pragmatically reuse it on another.

3. *Developers reuse code to save time and improve quality.* The most popular reason for reusing source code was to save time (Table 4.2, R8). This was repeatedly supported in the written questions with comments such as "reusing code is quicker and easier than [starting from scratch]". Several developers stated that they wanted to "leverage existing testing". Source code was more desirable as a candidate to reuse if tests existed for it, as these tests increased the developers' trust in the quality of that code. Other popular reasons for performing this type of reuse task included, "When I need the functionality but cannot call it directly", "code represents intricate functionality that cannot be written from the ground up without introducing errors", "grandfather[ing] old features into new versions of an application", "UI [User Interface] consistency", and to "gain insight into what parts can be abstracted into reusable components."

4. *Developers want to understand a feature's dependencies.* Reasoning about source code, especially code someone else has written, can be very difficult. Our subjects agreed that keeping track of the facts relevant to a reuse task while navigating the source code was difficult (Table 4.2, R13). Specifically, identifying the dependencies of the code they wanted to reuse on its original system was of importance (Table 4.2, R14). One respondent indicated that they were "less likely to reuse code that is tightly coupled to its context"; the rationale for this was that tight coupling makes the code "difficult to understand; it is hard to reduce the dependencies [on the rest of the system]". This was echoed by other respondents: "what are the dependencies

on other libraries?", "what other libraries will I have to pull in to use this code?", and "I need to know the dependencies; can I accept them into my project?".

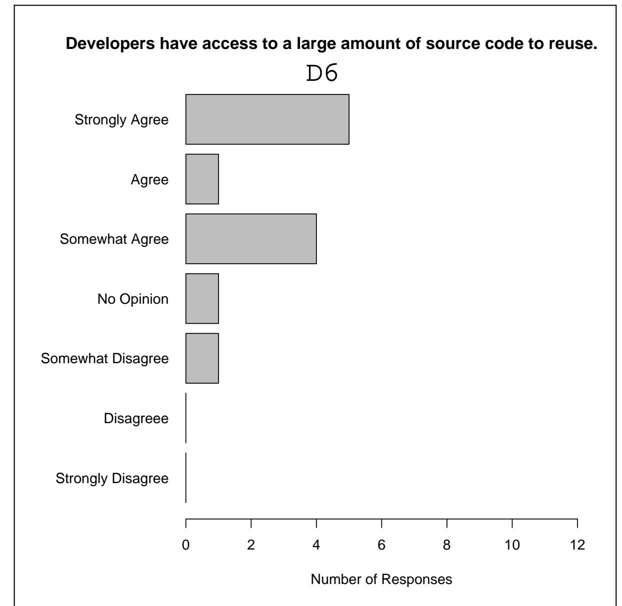
While attempting to understand a particular piece of code, many developers stated that they would sketch out its structure visually on paper. Several others wrote notes either on paper or as annotations within the code itself. Other developers would copy the source code from of its original context and into their system to see how it would "bleeds" (compilation errors are shown in red in many IDEs) in order to get a feel for how compatible the code fragment might be with their system, and how dependent the fragment was on its originating system.

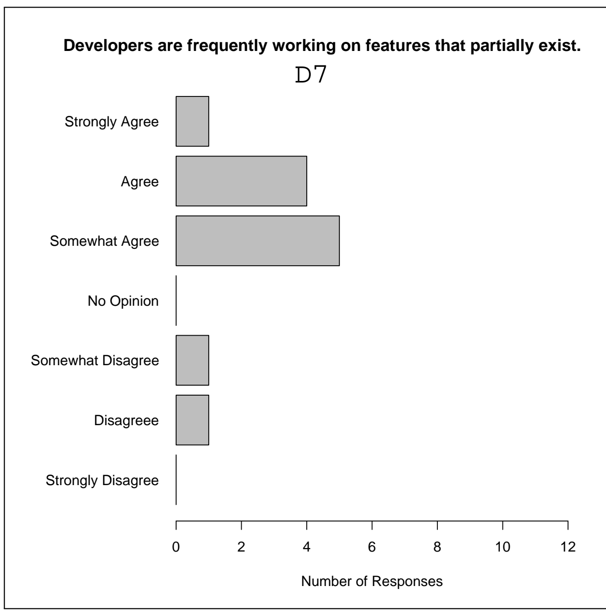
Key	Average	StdDev	Statement
D6	5.67	1.31	Developers have access to a large amount of source code to reuse.
D7	5.08	1.32	Developers are frequently working on features that partially exist.
R1	6.83	0.37	Developers are reusing source code.
R2	6.42	0.76	Developers are reusing whole classes.
R8	6.33	0.75	Developers reuse code to save time.
R11	2.75	1.30	Developers would prefer to write source code from scratch than reuse it.
R13	5.50	1.32	Developers find it difficult to keep track of all of the relevant details while navigating a program's source code.
R14	6.58	0.49	Developers want to understand a feature's dependencies before reusing it.

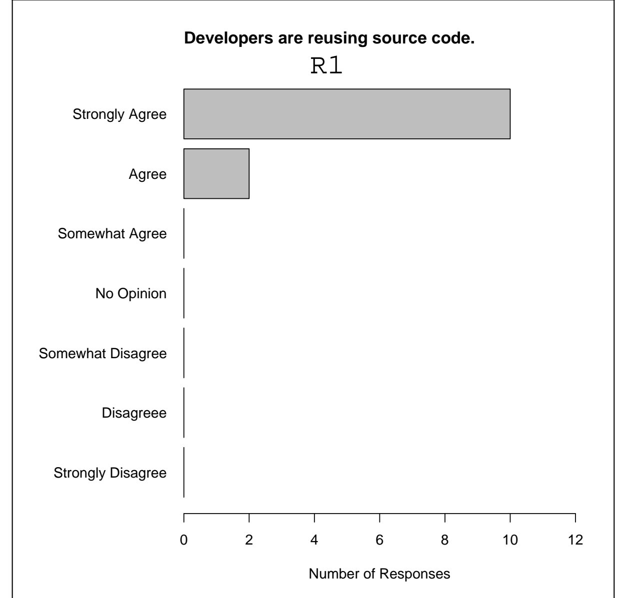
Table 4.2: Selected industrial pragmatic reuse survey results.

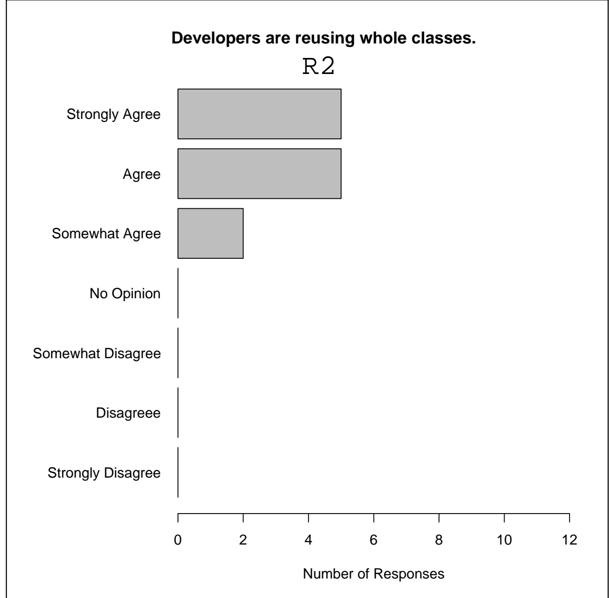
4.2 Other Findings

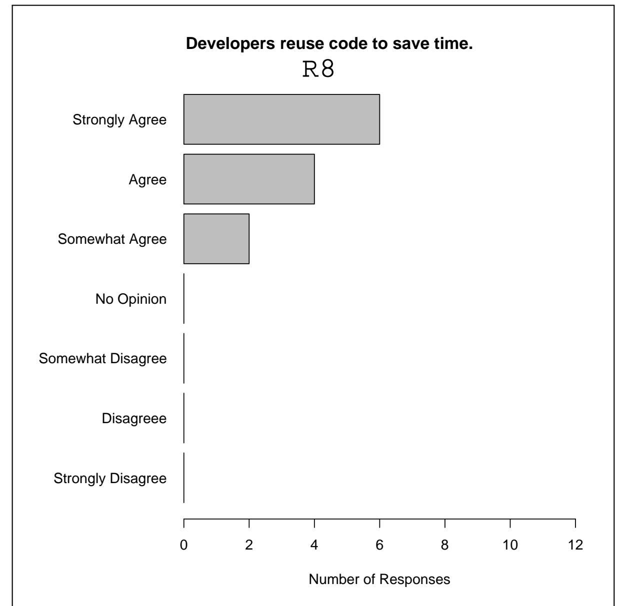
The respondents made several other interesting observations, in addition to those that support the four categories listed above. One respondent noted that "we glue pieces together, we don't write systems from the ground up (it's not the 80's anymore)". Another respondent mentioned that by reusing some source code, "the original author can improve their code in future iterations starting a mutually fulfilling cycle." Dependency understanding issues were clearly paramount to the respondents, "I must understand the dependencies", "visualizing the degree and exact relationships would be useful", "it would be nice to identify any dependent elements during cut and paste", and "highlight what's local and what's out of context when I paste." The respondents also offered several rationalizations for performing pragmatic reuse tasks, "quicker and easier to rewrite than to start from scratch", "if the API is not visible to me", "to leverage existing work (and testing)", "to reuse functionality between different applications", "to port between platforms".

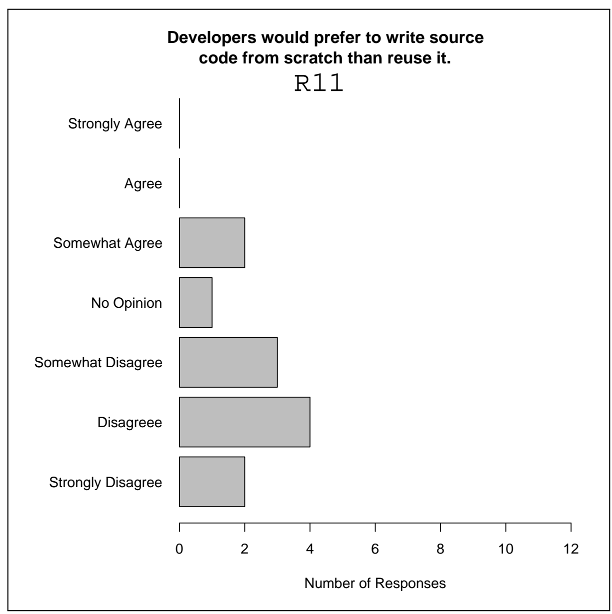

Developers have access to a large amount of source code to reuse.
D6
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses

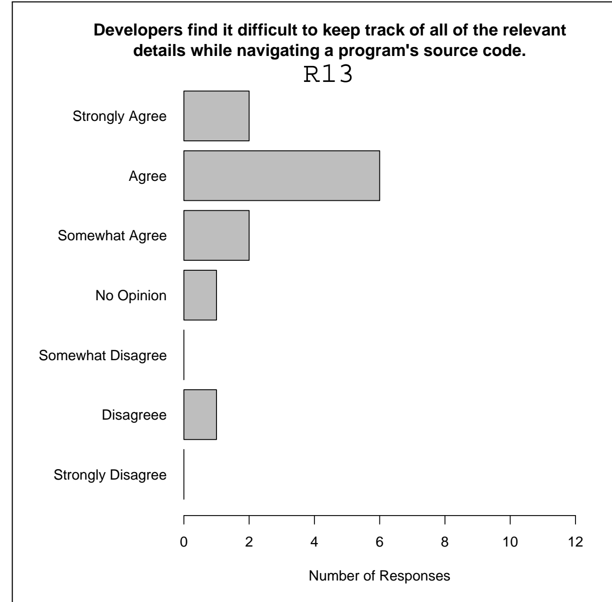

Developers are frequently working on features that partially exist.
D7
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses

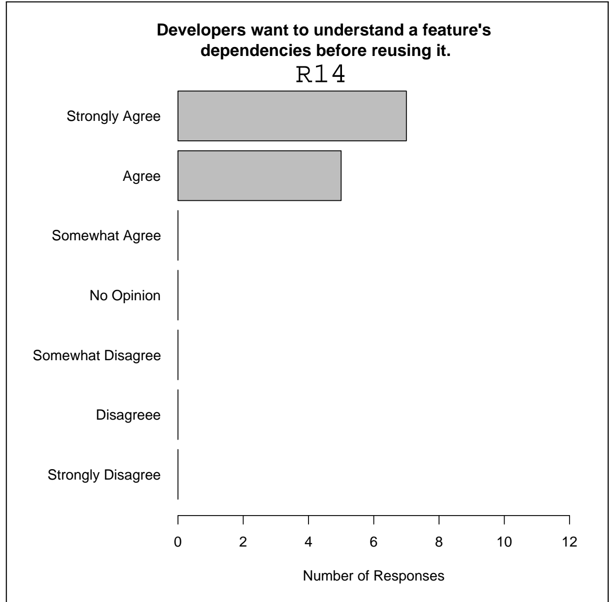

Developers are reusing source code.
R1
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses


Developers are reusing whole classes.
R2
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses


Developers reuse code to save time.
R8
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses


Developers would prefer to write source code from scratch than reuse it.
R11
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses


Developers find it difficult to keep track of all of the relevant details while navigating a program's source code.
R13
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses


Developers want to understand a feature's dependencies before reusing it.
R14
Strongly Agree
Agree
Somewhat Agree
No Opinion
Somewhat Disagree
Disagreee
Strongly Disagree
0
2
4
6
8
10
12
Number of Responses

4.3 Survey Limitations

Only 12 developers responded to our questionnaire; due to the limited sample size, the study does not indicate that all developers perform pragmatic reuse tasks but instead provides evidence that at least some developers do. As the number of developers who indicated in our study that they perform these tasks was high, we took this as enough validation to warrant further investigation.

One downside of performing a questionnaire-based study is that we did not have the opportunity to interview our respondents. This meant that the amount of information provided in response to the open-ended questions was somewhat less than we would have preferred. Because of this we resolved to ensure that future studies involved an in-person or telephone-based exit interview in order to get more comprehensive details from our participants.

4.4 Summary

The results of the survey confirmed for us that developers are performing pragmatic reuse tasks. It also confirmed our suspicion that the structural dependencies, within the source code that the developer wanted to reuse, played an important role in performing pragmatic reuse tasks. One higher-level observation that arose from the study was that while navigating the source code and trying to understand its dependencies, what the developer was really trying to do is gain an understanding of how hard the reuse task would be: would it be an easy 15 minute activity or would they have to invest several days in getting the code to work? This cost estimation was crucial for them to determine whether their time would be best spent performing the reuse task or creating their own version of the functionality.

The majority of the respondents had performed pragmatic reuse tasks in their industrial context; while they undertook these tasks for a variety of reasons, they ultimately believed that these reuse tasks were effective for their context. The wealth of interest in understanding the dependencies involved in pragmatic reuse tasks led us to further examine tool support for both understanding pragmatic reuse tasks as well as support for performing them.

Chapter 5

Pragmatic reuse model

At its simplest level, a pragmatic reuse task involves locating a set of structural elements from one system, extracting them, and reusing them within another system. These tasks embed within themselves a certain set of requirements; a model to encode these tasks must be able to describe the set of elements a developer wants to reuse and capture how they should be treated in a pragmatic reuse task. This section describes the requirements of a pragmatic reuse model as well as specific details pertaining to our concrete implementation of this model.

5.1 Model requirements

In order to encode a pragmatic reuse task, a pragmatic reuse model must support two primary design requirements. The model must be able to:

1. *Describe the elements involved in the reuse task.* The developer requires a mechanism to track the structural elements he has investigated while he is exploring unfamiliar source code. The model must be able to record any set of structural elements and the relationships between them.

2. *Capture the intent of the developer.* Describing the structural elements on their own is not very useful; being able to capture what the developer intends to do with these elements when he reuses them is of paramount importance. This requirement aims to transform a pragmatic reuse plan from a set of structural elements into a *lightweight specification* of a reuse task. By intent we mean that the developer must be able to tag a structural element with a specific decision relating to how the element should be treated in a reuse task. These decisions can be further augmented by additional pieces of metadata, as required. Each structural element can have only one decision associated.

In addition to these primary requirements, pragmatic reuse models have five secondary requirements; the model must be:

1. *Consistent.* Decisions associated with structural elements must be applied consistently across the entire model. For example, if the developer chooses to apply a specific decision to a field, every reference to that field must be treated equivalently; special cases must not be permitted. By being consistent the model is simpler to understand and encourages the developer to make system-level decisions, rather than instance-level decisions.

2. *Lightweight.* Development tools in general are fairly simplistic in their operation: most activities performed by developers in IDEs are transparent in their operation and simple in their goals. The model must not be overly complex; this requirement seeks to minimize developer frustration and learning curves. The model's contents must map to concepts developers already understand and interact with regularly.

3. *Precise.* It must be possible to traverse from the model's representation of any structural element to its declaration in the source code from which the model was populated. Furthermore, every location in the source code that is relates two structural elements must also be derivable from the model.

4. *Language independent.* The model should not be tied to specific language features of any particular language. This requirement ensures that the assumptions embedded in the pragmatic reuse model do not hamper the future applicability of the model to other languages. That said, the model can, and does, require the language to be object-oriented.

5. *Amenable to automated interaction.* The goal of the model is to capture the structural elements and their reuse intent both to help the developer plan a reuse task, but also to help them perform it. The model must be programmatically accessible to enable tools to interact with the model and the source code the model was generated from.

The primary design requirements are descriptive; they aim to help the developer describe the nature, extent, and intent of a pragmatic reuse task. In this way, the pragmatic reuse model is a blend of the developer's mental model, their understanding of the code they are trying to reuse, and a model of the task.

Some of the secondary design requirements are prescriptive; they aim to constrain how the developer can perform the task. In particular, the constraint that the reuse plan be consistent reduces the amount of flexibility a plan can have. Additionally, the lightweight nature of the plans limits the scope of the decisions a plan can encode. Both of these requirements were added for comprehensibility of the pragmatic reuse plan: we wanted to make it easier for a developer to understand their pragmatic reuse plan without having to consider scores of special cases and exceptions.

5.2 Concrete model implementation

This section details our concrete implementation of a pragmatic reuse model that meets the requirements listed above; the only exception is that our implementation of the model has only been validated with Java-based pragmatic reuse tasks. We call our implementation of the pragmatic reuse model a pragmatic reuse *plan*. This implementation of the pragmatic reuse model aims to encode as few direct facts as possible, enabling additional information, if it is needed, to be derived from the existing model. Our pragmatic reuse model maintains a list of zero or more structural elements that have zero or more pieces of additional metadata associated with each of them. Each structural element can have zero or more relationships to other structural elements; additionally, each structural element can be tagged with zero or one reuse intent decision. Pragmatic reuse plans describe three kinds of structural elements: classes, fields, and methods. Each element is identified by its fully qualified name. Each class has an additional attribute corresponding to its originating system; as Java does not permit namespace collisions within packages, the plan uses the fully-qualified name as a key to uniquely identify any structural element. Pragmatic reuse plans are encoded in a graph structure: each of the structural elements is represented a node in the graph. These nodes are linked by edges corresponding to the relationships between them.

5.2.1 Relationships that exist between structural elements

Pragmatic reuse plans encode five kinds of relationships between structural elements:

- *Inherits.* Inheritance relationships are recorded between classes and interfaces. A class can extend zero or one other class and can implement zero or more interfaces. An interface can implement zero or more other interfaces. Inherits relationships are represented as an edge between two classes or interfaces.

- *Calls.* Method call relationships are recorded whenever a method is referenced within a program. Method calls include calls to type constructors (e.g. `new Foo()`), static calls (e.g., `Foo.bar()`), and super calls (e.g., `super()`). Method calls generally involve by one method invoking another, although two special cases exist: (1) a field can instantiate a type, thus calling its constructor; (2) static initializers can both instantiate types and invoke methods on those types. A call is represented by an edge from the originating method to the target method; in the case of field initializers or static initializers, a placeholder class initializer method is synthesized to act as the originating method.

- *References.* Field references are recorded whenever a field is referenced in a program. Field references can occur both within methods and within static initializers and when fields are referenced. Like calls relationships, references relationships are represented by an edge between an originating method where the reference takes place to the target field.

- *Has-Type.* This relationship only exists for fields: the fully-qualified type of the field is recorded, in addition to its fully-qualified name. The has-type relationship is represented by an edge from the field to a class or interface.

- *Containment.* In addition to their names, structural elements have metadata recording their containment relationships. Fields and methods are contained by classes and interfaces, and classes and interfaces in turn are contained by packages. Containment enables elements and relationships to be aggregated from the model (e.g., find all of the calls made by all of the methods in class `Foo`).

  The containment relationship is different from the other relationships as it is implicitly encoded in the structure of the source files. Containment enables elements and relationships to be aggregated from the model (e.g., find all of the calls made by all of the methods in class `Foo`).

5.2.2 Decisions a developer can make about a structural element

Our model enables one of six primary triage decisions to be encoded for any structural element by the developer. The decision represents how the developer intends for the structural element to be handled during the reuse task.

- *Accept.* If the developer decides that the structural element he is investigating represents an element that he should reuse, he can tag that element as accepted. Methods and fields cannot be accepted without the classes containing them being accepted (or they would not have anywhere to be reused to); this decision was made to minimize the difference between the pragmatic reuse plan and the source code itself.

- *Reject.* Structural elements that represent functionality the developer does not want to reuse can be tagged as rejected. If a class is rejected, all of its contained methods and fields are also rejected; however, individual methods and fields can be rejected from an an accepted class.

- *Remap.* Any structural element deemed by the developer to be similar to an element that already exists in his system can be remapped; when the developer chooses to remap an element he must also specify the element in his own system he would like the element to be remapped to. This means that the element itself will not be reused or rejected, rather any reference, call, or inheritance relationship to the structural element will be changed from the element itself to the element the developer specified.

- *Already Provided.* Common structural elements between the developer's system, and the source system he is investigating reusing source code from can be considered common. Common structural elements often arise when the source and target system are dependent on the same libraries or frameworks; in these cases the structural elements do not need to be reused to

satisfy the dependencies of any accepted code as they already exist in the developer's target environment.

- *Extract.* This special case of remapping applies only to fields. A field can be reused without its containing class by extracting it from that class and inserting it into one of the classes the developer has specified should be reused. This type of decision is best suited for reusing constants. This kind of decision was added based on feedback from industrial developers (this is detailed in Section 6.3).

- *Inject.* Another special case added at the request of industrial developers was the ability to inject any arbitrary fragment of code into a class they were reusing. They wanted to do this so they could add their own stub to the source code that they could then remap fields and methods to point to. This was to account for cases where they did not want to reject an element but did not have a suitable element within their own system to remap the element to.

Each of the developer's decisions will have implications on how much the reused source code has to be altered when the reuse task is performed. Any edge from an accepted structural element to a rejected structural element will result in a dangling dependency when the reuse task is performed; a developer could introduce many dangling dependencies into the reused code by rejecting even a single element if it is frequently called or referenced from the source code he has accepted. In general, these dangling dependencies that arise from relationships to rejected elements represent code the developer does not consider to be central to the reuse task; because of this, these dangling dependencies represent inheriting, referencing, and calling relationships that the developer will remove from the source code he has reused. Similarly, edges originating in accepted elements to remapped elements will also result in dangling dependencies; however, unlike with rejected elements, these dangling dependencies will not be removed but changed to reference the alternative structural element specified by the developer in the reuse plan. References to extracted fields from accepted elements will also be updated to point to the extracted field's new location.

5.3 Pragmatic reuse plans as lightweight specifications

Pragmatic reuse plans contain much of the information needed to undertake a pragmatic reuse task; as such, these plans represent lightweight specifications of pragmatic reuse tasks. Unlike formal specification schemes such as Z notation [Abrial et al., 1980] or B-Method [Abrial, 1996], pragmatic reuse plans are not provably correct with respect to their given specification.

Within the context of a pragmatic reuse task, being provably correct is an overly burdensome requirement as the overhead required to create a formal specification of the task may overwhelm the benefits of performing the task. Pragmatic reuse plans relax the restriction of being correct in exchange for a drastic reduction in their complexity and the time it takes to create them.

Our intention is not to fully automate pragmatic reuse tasks but to enable developers to be more effective when they are performing them. While pragmatic reuse plans are not necessarily complete, they are amenable to lightweight validation schemes. For instance, any edge from an accepted structural element to a structural element that has not been tagged with a decision can be easily detected from a pragmatic reuse plan; a tool could recommend that a developer investigate these structural elements in order to reduce the ambiguity in their plan. Pragmatic reuse plans are not positioned to supplant formal specifications, rather they are trying to fill a niche where these specifications would impose unnecessary complexity on the developer.

5.4 Relationship to the concern graph model

Several of the requirements of the pragmatic reuse model are similar to those given by Robillard for his Concern Graph Model [Robillard, 2003; Robillard and Murphy, 2007]. While Concern graphs and pragmatic reuse plans contain the same structural elements that have the same relationships between them, their intent diverges significantly. Concern graphs aim to bring together a set of structural elements to aid program understanding approaches; pragmatic reuse plans aim to extend this model to not only aid comprehension but to encode actions (via decisions) that can be used to actually perform specific tasks. Thus, pragmatic reuse plans encode the developer's intent within the context of a specific kind of task; concern graphs do not.

5.5 Structural analysis

Our model is populated by the static structure of the source and target systems. The static structural analysis engine used in this dissertation was directly derived from the Strathcona Example Recommendation System [Holmes et al., 2006]. Our static analysis engine had two modes, one for extracting data from the project hosting the code that was being reused (the source project) and one for extracting data from the developer's project (the target project). For the target project all that was extracted was the structural element names for every interface, class, method, and field in the system; while containment relationships were still inferred, no other structural relationships were extracted. For a source project, the static analysis engine would extract all of the structural elements as for the target project but it would also extract all of the structural inheritance, calls, and references relationships from the source code.

The primary limitation of our static analysis technique is that we were unable to accurately extract polymorphic type information. That is, if the source code had this snippet `IFoo foo = obj.getFoo()`, we would infer that `foo` was of type `IFoo` even though at runtime `foo` may have been of type `ConcreteFoo`. The analysis engine would however identify the inheritance relationship between `IFoo` and `ConcreteFoo` and present this in the model.

The structural analysis system takes both the source and target systems in order to determine the common structural elements between these projects. Currently, we assume that there are no fully-qualified name collisions for structural elements that are not in the `default` Java package. If a fully-qualified name from the target project matches one from the source project they are considered equivalent.

5.6 Summary

This chapter described the structural elements, their relationships, and the decisions that can be applied to them in our implementation of pragmatic reuse plans. The elements and their relationships are automatically extracted from the source code through structural analysis. The decisions about these elements are added by the developer as they progress through their pragmatic reuse plan. Pragmatic reuse plans describe the elements involved in a reuse task and capture how the developer intends to manage those elements when performing the reuse task; these plans are also consistent, lightweight, precise, language independent, and are amenable to automated interaction.

Our pragmatic reuse plans capture five structural relationships between packages, classes, interfaces, fields, and methods: inherits, calls, references, has-type, and containment. Developer intent can be encoded through six decisions: accept, reject, remap, already provided, extract, and inject.

Chapter 6

Graph-based pragmatic reuse planning

We built our first prototype of the Gilligan tool suite as a literal concrete realization of the pragmatic reuse model described in Chapter 5. This prototype focuses on the initial part of the pragmatic reuse process: helping developers to locate, to navigate between, and to understand the structural elements that they wish to reuse during their pragmatic reuse tasks.

6.1 Design goals

We had eight specific design goals when we designed the first prototype of Gilligan. The feedback we received from our initial industrial developer survey (Chapter 4) heavily influenced these goals; each feature of the prototype was designed to further one or more of these design goals (DG):

DG 1.1: *Provide an abstract representation of the structural elements and relationships being investigated.* One of the primary problems identified during the initial survey was that industrial developers found it difficult to remember all the details pertinent to their reuse task. We hypothesize that enabling developers to interact with an abstraction of the source code, rather than working through many editor windows and alternate IDE views, will provide a more cohesive experience that is easier for them to reason about.

DG 1.2: *Visualize all the structural elements and their relationships.* As many developers in the survey expressed a keen interest in the structural relationships of the source code they were investigating for reuse, our tool should provide a visualization of all of the structural elements, and their relationships.

DG 1.3: *Promote easy navigation between structural elements.* One of the most overwhelming aspects of following structural relationships in source code is having to open many different source code editors. We must enable developers to navigate their source code using an abstract representation to reduce disorientation.

DG 1.4: *Provide a high-level overview of the reuse task.* We would like to encourage developers to think about pragmatic reuse tasks from a higher-level, rather than getting lost in the very low-level details of the source code. To do this, we wish to provide a means to gain a global overview of the pragmatic reuse task.

DG 1.5: *Explicitly record decisions about structural elements.* The pragmatic reuse model can explicitly capture developer intent; our tool must enable developers to enter their reuse decisions into the pragmatic reuse plan.

DG 1.6: *Make the structural elements that have been triaged easily differentiable from those that have not.* Another problem with navigating through many source code editors is that while you can see which editors you have opened, it is difficult to see what portions of those editors you have examined and which you have not; this makes it more difficult for a developer to understand where they should investigate next. Our tool should clearly identify to the developer structural elements they should investigate before considering their plan complete.

DG 1.7: *Make performing pragmatic reuse tasks more systematic.* Navigating through unfamiliar code can be overwhelming and distracting. Prior research has listed the unsystematic nature of these reuse tasks as problematic(Chapter 3). As much as possible, our tool should aim to help developers proceed with their investigation as systematically as possible.

DG 1.8: *Conscientiously avoid work that does not directly aid in the completion of the reuse task.* Planning a pragmatic reuse task cannot take more time with our tool than it would take to perform manually, otherwise developers will never use the technique. As much as possible, the tool must be lightweight, and allow developers to integrate our tool into their workflow.

6.2 First Gilligan prototype

The main objective of the first Gilligan prototype was to help developers plan their pragmatic reuse tasks. Figure 6.1 contains an overview of how Gilligan fits into the overall pragmatic reuse process. Once a developer has located a system containing a feature they would like to reuse, they can search the source code using lexical tools for a starting point to begin their investigation. They can then use Gilligan to continue planning their reuse task from that point; with only a few clicks they can start the tool by selecting their source and target projects [DG 1.8]. Gilligan then extracts the structural relationships statically from the source code and stores them in the pragmatic reuse model for later retrieval. The developer can select their starting point in the source system from a list of struc-

tural elements in the system (through a tree view with a text filter) and start exploring the structural relationships in the system from there.

Gilligan helps developers to visualize and navigate the structural dependencies within the source code they are investigating; the end result of this investigation is a pragmatic reuse plan that the developers can use to determine if the plan is meritorious and should be performed or if the plan should be discarded. We hope that by making a pragmatic reuse plan, the developer will be making these decisions for rational technical reasons. This is in contrast to dismissing a reuse task out of hand for seeming like a bad idea, or pursuing a poor pragmatic reuse task without knowing that there are significant technical hurdles that may make it impractical. If the developer decides that the reuse plan is worthwhile, they can use the plan as a set of instructions guiding them through enacting their plan by copying and modifying the source code they have identified for reuse.

The results of the prototype and evaluation discussed in this chapter have been previously published [Holmes and Walker, 2007a].

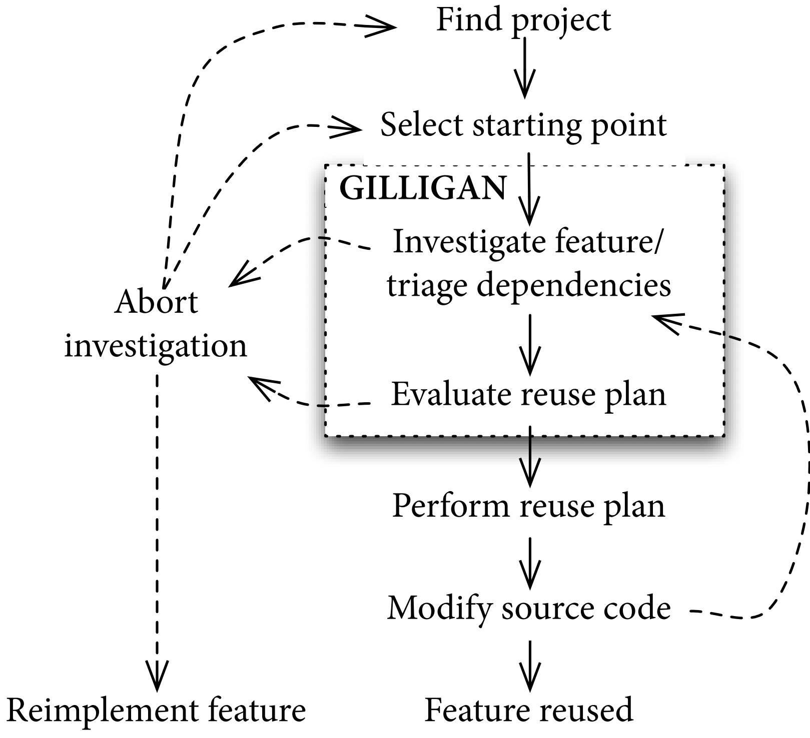


Figure 6.1: Gilligan's role in the pragmatic reuse process.

Gilligan was implemented as a set of bundles (formerly called plug-ins) for the Eclipse IDE.[1] Figure 6.2 provides a screen capture of Gilligan as it would look at the outset of the scenario described in Chapter 2. This prototype of Gilligan directly displays the contents of the pragmatic reuse model

[1]http://eclipse.org v3.0.1 to v3.4.1

for the developer to investigate; it relies on a graph-based metaphor: structural elements are represented as nodes, the relationships between elements are represented as edges. Nodes are decorated according to their kind (package, class, method, field) with a small icon on the leftmost side [DG 1.1, 1.2, 1.4]. Table 6.1 lists the various icons and their associated meanings; these icons were sourced from within Eclipse itself to keep the icon scheme consistent with what Eclipse developers see regularly. The edges Gilligan uses to represent relationships are shown in Table 6.2. The colours of these edges vary depending on whether a node adjacent to the edge is selected or not. In their default state, inheritance relationships are blue, calls and references black, and containment grey. When selected, all edges change from a width of 1 pixel to 4 pixels. Contains edges and inheritance edges are solid while calls and references edges are dashed.

Icon	Description
	Package
	Interface
	Class
	Method
	Field

Table 6.1: Standard Eclipse icons used by Gilligan.

Edge	Description
- - ►	Call or reference
─▷	Inheritance
─〉	Containment

Table 6.2: Kinds of edges used by the Gilligan graph-based representation.

This prototype supports all the decisions listed in Section 5.2.2, with the exception of field extraction[2]. Decisions are made by right clicking on a node in the graph-based view and selecting the decision that is appropriate [DG 1.5]. These decisions are annotated with bright colours; these colours are given in Table 6.3. We chose to represent the decisions with bright colours to enable developers to glance at their reuse plan and instantly get a sense of how much code they were reusing, rejecting, remapping, and how much was common [DG 1.4, 1.6]. This at-a-glance support is intended to help developers to feel on top of their task instead of feeling that they are mired in the details.

[2]This support was added in the second prototype based on the feedback we received from this prototype.

Colour	Associated Decision
(green)	Accepted structural element
(red)	Rejected structural element
(blue)	Remapped structural element
(yellow)	Common structural element

Table 6.3: Gilligan colour scheme for encoding pragmatic reuse intent.

Gilligan provides one complete perspective[3] consisting of four primary views.

- *Gilligan Reuse Graph.* This view exists in the top-left hand corner of Figure 6.2; this view maintains a list of all of the structural elements that the developer has investigated and are present in the exploratory view. The Reuse Graph view has four key features: (1) it explicitly encodes the containment relationships between structural elements through its tree structure; (2) it enables developers to view similarities and differences in their decisions for similarly contained structural elements as the label background in the tree is coloured with the decision colours [**DG 1.6**]; (3) it allows quick searching with the text filter at the top of the view, this enables the developer to search using both plain text and regular expressions [**DG 1.3**]; and (4) it is linked to all of the other views, if the developer selects a node in the tree it is instantly selected in the graph view and its details are shown in the properties view [**DG 1.3, 1.4**]). Only containment relationships are shown in this view, the developer must use the Exploratory View to see the structural relationships between elements.

- *Gilligan Exploratory View.* This is Gilligan's core view (large panel at the top of Figure 6.2). It allows the developer to navigate all of the structural elements in the system by traversing their relationships from element to element [**DG 1.3**]. This view provides a graphical representation of all of the structural elements and their relationships in the system [**DG 1.2**]. One of its primary means of scaling is by only showing new elements as a developer asks for them; for example, when the developer starts Gilligan with their initial location in the source code, only a single method, class, or field is listed in the graph. As they expand nodes their relationships are shown [**DG 1.7**]. In this way, the exploration is similar in nature to working through the source code: the developer only sees the details they ask for; however, unlike the source code the developer is able to stay in the same view all of the time and are able to maintain the context of how the element they are investigating interacts with the rest of the system [**DG 1.8**]. As the developer opens new structural elements any relationships to previously open structural elements are shown. The view provides a number of interaction techniques for the developer:

[3] In Eclipse lingo, a perspective is a cohesive set of individual views that are linked, interact, and work together for a common task (e.g., debugging is one of the Java perspectives).

- *Hover.* When the developer hovers their pointer over a structural element the edges and adjoining nodes are highlighted (and their borders and edges made heavier-weight) in response [**DG 1.3**]. This feature was added to make the structural relationships the developer was currently investigating more obvious when the graph grew to be large. The node that the mouse is over is highlighted in red; edges are highlighted in red as well while adjacent nodes are highlighted with a yellow border. The effects of the developer hovering can be seen in Figure 6.2 and Figure 6.3; in both figures, the developer has hovered over the `drawChart(..)` method. In Figure 6.2 only one the parent of `drawChart(..)` is related to the method as the developer has yet to explore the `drawChart(..)` method. In Figure 6.3 the developer has further explored the task so the highlighting shows many more relationships.

  While the developer is hovering over an element a small tooltip-like panel appears below the node giving the developer quick access to several major pieces of functionality. This panel has a small square button the colour of each decision the developer could make, as well as a button to view the source code corresponding to the structural element in the source code view [**DG 1.5**]. This panel only stays active if the developer moves their cursor onto it while hovering; otherwise it disappears quickly so the developer can concentrate on the default hover highlighting.
- *Click.* If a developer clicks on an element, it becomes selected. The hover highlight remains active until the developer either clicks on, or hovers over, another element. The developer can move any node after they have clicked on it; the edges to adjacent nodes will automatically update as the developer drags the node to its new location.
- *Right click.* By right clicking on the node the developer can bring up the same features that were available to them in the hover view, but in a context menu format [**DG 1.5**]. Each label in the context menu is preceded by a small icon that gives a graphical clue (e.g., colour or picture) that the developer can select without reading the label's text.
- *Double Click.* This action is the main way for the developer to continue exploring the dependencies in the system. When they double click on an element all of the dependencies of that element that are not already on the graph area added to the canvas [**DG 1.3**]. Gilligan utilizes a radial layout system as this was found during our initial informal experimentation to be an effective way to visualize structural elements.[4] As new elements are opened the old nodes are moved outwards as necessary to make space for the newly opened elements. To reduce disorientation they are moved outwards symmetrically from the middle of the element the developer opened; node opening is also animated in an effort to reduce disorientation caused by moving nodes.

[4] We investigated several layout algorithms while making this choice; four specific examples we investigated are shown in Appendix B. We decided the final example, a radial layout, was best suited to this project.

As nodes are selected in the exploratory view they are also selected in the tree in the left node; this means that the developer can always glance to see containment information, related decisions, and any information that is in the property view [**DG 1.4**]. Multiple selection is not allowed from either the graph exploration view or from the tree view.

- *Properties View.* The developer can get additional details about any node they have selected using the properties view (lower left corner of Figure 6.2). The standard fields in this view are name, declared in (contained by), state (decision), and comments. The name and declared in fields are read only but the developer can modify the decision through a drop-down menu; they can also add a comment if they think that there is some specific detail they want to remember that they would like to have stored with the model.

- *Source View.* The source code view (the bottom panel in Figure 6.2) allows the developer to look at the source code for any node they are investigating. This panel can be triggered from both the reuse graph and from the exploratory view. It is essential to not try to hide everything behind the abstraction, as developers are used to working with source code; without being able to access the source code directly they simply find indirect ways to do it and become frustrated (research has found that developers trust source code over any other artifact available to them [Singer, 1998]) [**DG 1.8**]. The source code is essential for reasoning about specific, important decisions, assessing the quality of the source code, and working with source code that does not have descriptive identifiers. Gilligan heavily augments the source code view by decorating the area behind specific textual elements with the colours corresponding to the developer's decisions; for example, if the developer has rejected the method `methodCall()`, accepted an interface `IListener` and remapped the field `MyClass.FieldA`, they would appear as `methodCall()`, `IListener`, and `MyClass.FieldA` for any instance of those elements in the source file the developer is perusing [**DG 1.6**]. These text colourings were added to reenforce the decisions that the developer made in the abstract model and to project these onto their concrete realization in the source code. In particular, having the yellow (already provided) annotations on the source code was extremely informative when projects used several common libraries and frameworks; this allowed the developer who was looking at the source code to ignore a large portion of it directly, without even needing to investigate an element only to discover that they did not need to consider it.

To help manage the complexity of the graph, nodes can be collapsed into their parents. This collapse functionality simplifies the graph by eliding details the developer is no longer interested in seeing (such as collapsing methods into their parent class, or a class into its package). Some nodes in Figure 6.3 have been collapsed into their parent packages. Developers can collapse nodes either through a right click action or via the tooltip panel. This functionality was added to Gilligan to help it scale to larger reuse plans.

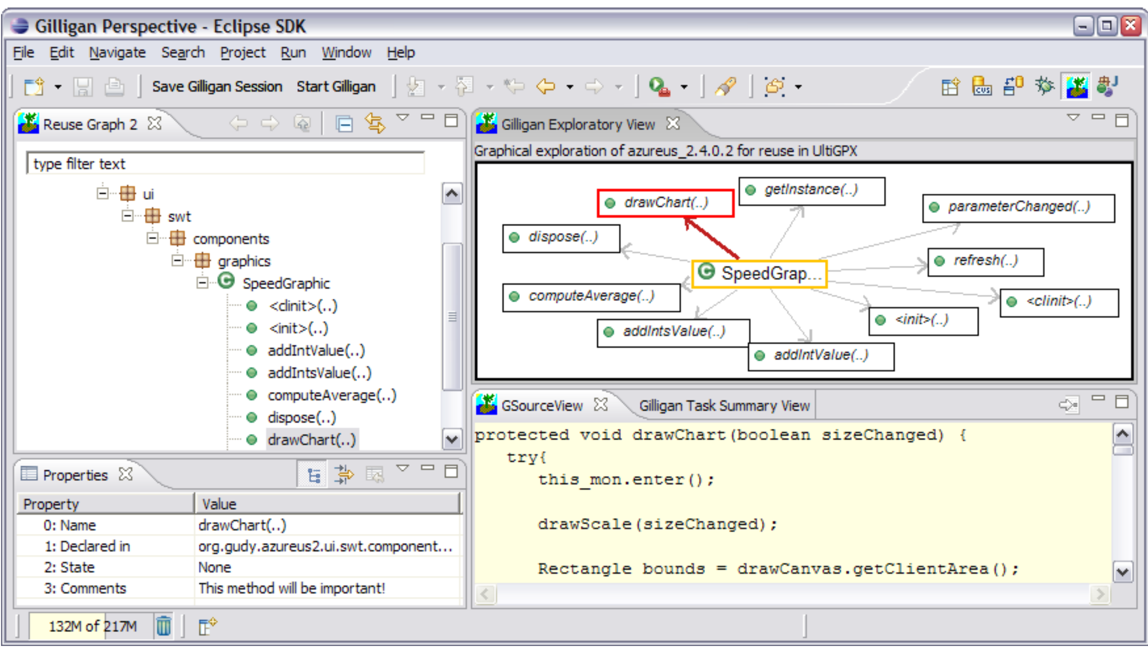


Figure 6.2: Screen capture of the graph-based Gilligan prototype.

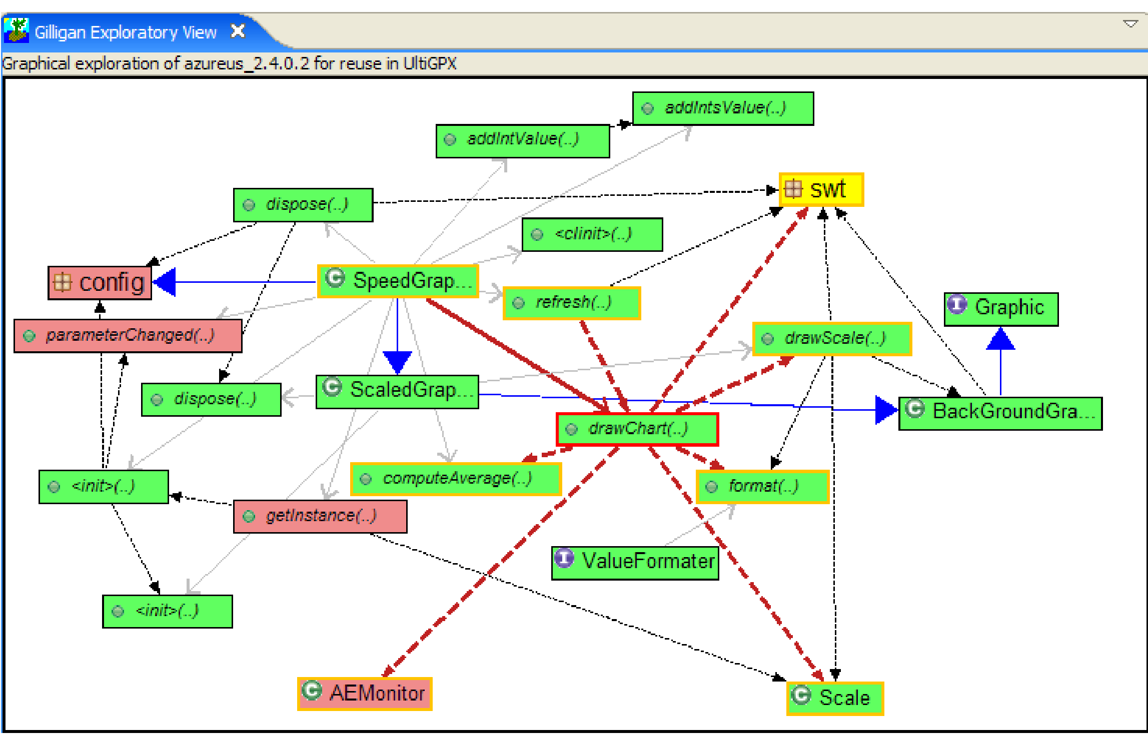


Figure 6.3: Azureus graphics feature reuse plan.

6.2.1 Using the graph-based Gilligan prototype

Thus far we have described how the developer interacts with Gilligan, but not how they actually *use* it to create a pragmatic reuse plan. After selecting their starting node and being presented with an initial set of nodes, developers investigate nodes that seem interesting to them. If there are many yellow, already provided, elements the developer can simply collapse them so they are not interfering with their exploration of the task. As they see nodes that they are interested in they can double click to follow the structural dependencies as deep into the source system as they would like; once they have seen enough they can just select another node and continue on in a new direction. Developers are encouraged to make decisions about the nodes they investigate as they proceed through the dependencies. Nodes that have been tagged with a decision are shown in bright colours; by tagging nodes, developers can look for blank nodes as indications of where they should investigate next. Figure 6.3 shows what a more complete reuse plan, in which many nodes have been tagged, looks like. Gilligan does not impose on the developer the order in which they should investigate nodes although by directing them to nodes they have not investigated and making it visually obvious which nodes they have investigated, the developer can maintain a sense of forward momentum without feeling overwhelmed [**DG 1.4, 1.7**]. One easy way for the developer to feel like they are "done" investigating a reuse task is by noting that all the nodes in their pragmatic reuse plan have been tagged with decisions; in this case the plan can be considered complete.

6.2.2 Evaluating the reuse plan

Gilligan helps the developer focus on those dependencies that matter, recording his decisions about those dependencies to minimize the re-viewing of code fragments. Using the graphical view, it is visually evident which nodes need to be further addressed before the investigation is complete. While this systematic process helps the developer to see what decisions he has made, it does not impose any particular order in which those decisions must be made; the developer can iterate on the graph in any way that is appropriate to his task.

At any point during the investigation of the feature, the developer can evaluate the state of his reuse plan. The developer may notice early on that there are far too many rejected and remapped dependencies to easily reuse the feature. If the developer completes the reuse plan, he can use its structural description and annotations to make an informed decision about reusing the feature.

6.2.3 Application to motivational scenario

To ensure that this Gilligan prototype could successfully plan a reuse task we applied our tool to the scenario described in Chapter 2. We started by selecting the Azureus project from our workspace in Eclipse[5] as the original project of the feature we wanted to reuse and the UltiGPX project as the target where the code was to be reused. As mentioned in Chapter 2 `SpeedGraphic.drawChart(..)` was

[5] http://azureus.sf.net (v2.4.0.2)

the starting point for this functionality so we set this as the starting point in Gilligan. Gilligan's initial visual state is shown in Figure 6.2. We navigated the `SpeedGraphic` class and its dependencies to identify those portions of the class that were relevant to the graph drawing feature that we wanted to reuse while identifying any other Azureus dependencies we did not want to reuse.

Opening the dependencies for the `drawChart(..)` method, we found 21 structural dependencies; however, 14 of these (from 4 classes) had been automatically coloured yellow by Gilligan—both Azureus and UltiGPX use the SWT framework. To reduce the clutter on the screen, we collapsed all of the `org.eclipse.swt.*` dependencies into a single `swt` package. The remaining 7 dependencies are from four different classes within Azureus. Double-clicking on the call edge to `AEMonitor.enter()`, we were presented with `drawChart(..)`'s source code in which the references to `AEMonitor` were highlighted. From this annotated source view we could see that `AEMonitor` was concerned with the locking functionality within the core of Azureus. This was not a feature we cared to reuse so we collapsed `AEMonitor`'s methods into itself and marked it as rejected. Another 15 minutes of investigation resulted in the reuse plan shown in Figure 6.3.

Using Figure 6.3 to estimate the effort required to perform the reuse task, we can see that we would have to reuse 5 complete classes and 1 partial class. We were going to have to manage 4 dependencies on source code (involving the Azureus network locks and configuration) that were rejected from the reuse task. Five classes in the `swt` package were common between Azureus and UltiGPX; no action was required of us to satisfy the dependencies on these classes. From this sketch, we can see that it should not be too difficult to extract this feature and integrate it with UltiGPX. Performing the extraction and integration—and following the plan—took less than an hour. The reuse plan helped in accomplishing the task, as whenever a compile error was encountered in the reused code fragments, we could check the reuse plan to see how we should manage it; this helped direct our integration activity. The speed with which we could plan and complete this task constituted further evidence that Gilligan was worth investigating further.

6.3 Evaluation

As this was our initial prototype for a reuse planning environment, we sought to address two high-level research questions:

1. Can industrial developers create pragmatic reuse plans using Gilligan?
2. Do these developers perceive any derived benefit from this activity?

The goal of this evaluation was not to "prove" Gilligan's effectiveness but rather to get feedback on our prototype and to gain insight into industrial developers' attitudes towards the tool support we had built. We gave the tool to four industrial developers working in different companies. We identified these four developers during our survey (Chapter 4) as they all work with the Java programming

language within the Eclipse IDE, the same environment supported by the first Gilligan prototype. In addition to verifying that the developers could plan reuse tasks, we wanted to know if the developers felt they could tackle larger reuse tasks with the tool than they would normally attempt. The four developers applied our tool to their own industrial reuse tasks and filled in a short questionnaire about each task they tried (this questionnaire is included in Section A.2 on p. 146). Gilligan was also instrumented to record the developers' navigation and decision actions as they were investigating their reuse task.

6.3.1 Case study 1

The first developer undertook two tasks: he extracted code from the open-source Standard Widget Toolkit (SWT) graphical framework for parsing both BMP and PNG image files. He wanted to reuse these pieces of code because they involve processing complex binary file formats that he did not want to have to write himself and he was unable to reuse all of SWT (which comprises 68 [**DG** [**DG**]]spread across 458 classes[6]).

6.3.1.1 BMP extraction.

The developer started Gilligan with the `WinBMPFileFormat` class as the initial node. Using the graphical view, the developer was able to quickly reject several methods in this class as they pertained to the writing of these files, which he was not interested in. Further exploration led him towards `LE-DataInputStream` and `ImageData`. He reused the former in its entirety, and just the data structure from the latter. In the end, he reused 497 lines of code and had no latent dependencies on SWT. Of the 14,081 possible nodes in SWT, the developer only visited 60. His final view of the feature had 27 visible nodes. Of the nodes he visited, he accepted 38, rejected 16, remapped 2, and 4 were already provided.

The developer then demonstrated how he would have undertaken the task manually. First, he copied `WinBMPFileFormat` into his new project. He then went down the list of compilation problems (there were many) and dealt with them individually. Any dependency he could not easily manage he left until later. At the end he went through the remaining difficult dependencies and also copied `LEDataInputStream` and the `ImageData` data structure into his workspace. Once the compilation errors were resolved he was done. His methodology was similar to what our tool provides: he used the compilation errors as markers for structural dependencies that were not satisfied within his target environment. Unfortunately, doing this manually forces the developer to undertake the task without having first determined that it is feasible. For example, if there was a dependency within the BMP code that required a specific dependency within the SWT framework the developer could not reuse, the developer would not have discovered this until they located the error corresponding to this dependency.

[6] http://www.eclipse.org/swt/

6.3.1.2 PNG extraction.

The developer began his investigation with the `PNGFileFormat` class. He was interested in immediately noting all of the class-level dependencies of this 471-line class. Unfortunately, the first Gilligan prototype tool was designed to support a bottom-up investigation style and he had to open `PNGFileFormat`'s methods to see these dependencies[7]. After opening these dependencies he had 92 nodes on the screen and had discovered that there were at least 20 classes of interest to him for this task. During this investigation the developer was interrupted multiple times by questions from his co-workers. After these interruptions he was able to go back to the visual plan to remember where he was; because his decisions were noted on the plan, these distractions did not cause him to go back and re-evaluate any nodes. With 92 nodes visible, the developer indicated he would appreciate being able to filter the nodes based on their type (for instance, only show class nodes) in order to make it easier to understand. In the end, the developer marked 20 classes as accepted, 2 as rejected and 1 as remapped.

When the developer actually did this reuse task he reused 23 classes (comprising ∼3 kLOC). During this reuse task he decided to change his mind about three decisions: he accepted one previously-rejected class, he changed the remapped class to accepted, and accepted one class he overlooked during the investigation. These changes were made primarily due to the complexity of the task he was pursuing.

In the post-task questionnaire the developer stated, "After trying to reuse the SWT BMP decoder, I wasn't convinced additional tooling was necessary (300 lines reused, fairly isolated to one to three classes). After trying to reuse the SWT PNG decoder, I changed my mind." He also indicated that he strongly agreed that Gilligan could help him "attempt larger, more complex reuse tasks". However, this case highlighted the need for further refinement of the user interface. While Gilligan did initially help the developer identify those types that were relevant to his task, he was eventually overwhelmed and had to use a hybrid approach that used both Gilligan and the manual techniques demonstrated in the last case.

6.3.2 Case study 2

The second developer wanted to reuse a feature that serialized his Java objects into XML so they could be sent over the network. This feature needed to be reused because the originating project was no longer being maintained. The developer started with a class that he knew was involved with the XML serialization functionality, so he explored its dependencies. He accepted 4 classes, remapped 2, and found that 8 were already provided within his target system. During his investigation he investigated 84 nodes (33 of which were visible in the final plan), accepting 13, rejecting 2, and remapping 4, while 18 were automatically marked as common. This reuse task took approximately 2 hours and in the end ∼900 lines of code were reused. The reuse task was a starting point for a refactoring task to make

[7] This shortcoming was addressed by our second prototype (Chapter 7).

the old feature conform to the new system. The developer found that the tool "helped me visualize the scope of reuse tasks and how much I would be able to reuse and what I would have to write."

6.3.3 Case study 3

The third developer wanted to reuse the virtual file system from a third-party application and to add this functionality to his own system. The reuse task involved reusing 9 classes and remapping two of the classes to equivalent functionality already provided within his own system; this task involved reusing ∼3000 lines of code. The developer investigated 49 nodes in the visual view, with 32 of them remaining in his final reuse plan; this investigation took 25 minutes. He accepted 25 nodes, while 5 were already provided. He also marked two nodes for remapping; he wanted to connect these nodes, corresponding to logging functionality, to those within his own application. During his investigation, he also investigated the source code for 12 specific method calls. This developer found that Gilligan helped him decide that this task was possible before carrying it out; however, he wanted the tool to clearly highlight which method-nodes had external dependencies on them. He also wanted the ability to hide nodes such as those representing already provided functionality.

6.3.4 Case study 4

The fourth developer also completed two tasks. In his first task, he wanted to reuse an implementation of an old feature within a new system; however, he needed to modify this implementation in ways that were inconsistent with the old application. In this task, he visited 42 nodes, with 24 ultimately being of interest to him. Of these, he marked 12 as accepted and 5 as remapped; 15 were automatically marked as already provided within his target system. The task ultimately reused only 200 lines of code and took only 20 minutes to accomplish.

In the second task, the developer attempted to reuse the GraphML parsing code from the Jung open source project.[8] This was a complex task that involved over 2000 lines of reused code. During the investigating phase, the developer identified the need for a "verify plan" feature for the tool; this feature would check one's accepted nodes and confirm that they do not have any non-triaged dependencies. He requested this feature because during this task he investigated 72 nodes (of the 23,157 nodes in Jung) which he found to be somewhat overwhelming to keep track of. He noted that the tree view was especially important for tracking large reuse tasks as the entries in this view are also coloured with the developer's decisions. In this task he accepted 10 nodes, rejected 4, remapped 5, and 13 were marked as already provided.

[8] http://jung.sf.net

6.4 Discussion

After the case studies, each of the four industrial developers agreed that Gilligan helped them plan larger reuse tasks than they would usually attempt; indeed, the tasks the developers attempted were all larger than the typical sizes identified in our initial survey. Of our stated goals, the developers responded most strongly to the aspects of Gilligan that supported DG 1.3, 1.5, and 1.6. These broke down to three key savings, from their perspectives: (1) they did not have to hunt through a maze of editors to navigate a dependency chain; (2) they did not forget the decisions they made during their investigation, nor did they have to try to remember the structure of the system or write it down; and (3) by seeing their decisions explicitly recorded they were able to direct their investigation to structural elements they had not yet visited and could avoid revisiting nodes they had already investigated.

The largest and most consistent piece of feedback about the planning interface we received was that the graphical approach started to degrade for larger reuse plans. Developers also found that it was awkward to work with the graph-based representation; their tools are mainly editor-, list-, and tree-based and switching to a graphical representation made them work harder than they expected. Based on this feedback, we worked extensively with several of these developers to brainstorm what pieces of information they needed most and to come up with a new prototype interface for planning pragmatic reuse tasks. The single fact that surprised us the most from these discussions is that the developers were mainly interested in the outgoing dependencies for any element they were looking at. Interestingly, this strongly contradicts DG 1.2 which involved showing all the structural dependencies to the developer. We resolved to rethink our planning interface with a more use-case oriented approach for the next iteration. This will be discussed in greater detail in Chapter 7. While we might have attributed some of this feedback to the fact that these developers were using the tool without formal instructions about how it operated, our own experiences using Gilligan strongly reinforced the view that the graphical prototype poorly scaled for planning larger reuse tasks.

In this evaluation, we also learned of two new decisions that our industrial participants wished they could have made while triaging structural elements. First, they wanted the ability to extract a field from its declaring class and to reuse it within one of their own classes without its declaring class. This was specifically to enable them to reuse constants more easily. Second, our industrial participants wanted the ability to "inject" a source code fragment into any class they had reused. By this, they meant the ability to add arbitrary methods or fields to their reused class. This was to enable them to add their own code to the source code they were reusing and then remap any field or method to this stub.

Two small features that were mentioned by several developers as being helpful were the source code view and the automatically tagged common dependencies. Developers liked being able to look at their decisions in the source code text itself, although they wished the colouring was applied to an editor they could change themselves instead of to a static text view. Gilligan's automatic tagging of common dependencies was also helpful to developers; they liked knowing they could not worry

about specific dependencies without looking into them at all.

DG 1.8 was not entirely fulfilled. While developers were able to create pragmatic reuse plans, and thought they could attempt larger tasks by doing so, working with an abstraction did not directly help them *perform* the task. Several times developers commented about additional features to help them verify their plan and actually perform the mechanical operations required to enact the reuse plan for them. We investigated means to address these concerns in our third prototype which is covered in Chapter 8.

Another interesting lesson learned from this set of case studies is that running controlled experiments is preferable for industrial developers. This is because actually getting useful information from them about tasks they performed on their own systems is problematic: most developers had to consult with their managers, and some had to discuss the matter with their company's legal department. These issues could have been avoided if we had provided the tasks ourselves. While this decision would negatively impact the external validity of our experiments, it would increase their internal validity; we decided this trade-off was acceptable to be able to continue to collect feedback from our industrial partners.

6.4.1 Limitations

These case studies involved only four industrial developers. While their six tasks were realistic for their context, we cannot claim their experiences will definitely generalize. This is a proof-of-concept that we used to check our progress with the prototype and to learn what future directions to take with the next version of the tool. While these developers, and their tasks, were varied enough to suggest that Gilligan is a valuable tool for planning and reasoning about pragmatic reuse tasks, more evaluation is required to both act on their feedback and further validate our approach.

6.5 Summary

Our initial Gilligan prototype tried to faithfully, and directly, represent pragmatic reuse plans through a graph-based visualization through which developers could explore and navigate the structural dependencies of the source code they were investigating for reuse. We had eight design goals for this initial prototype; upon evaluation, we discovered that one of these, DG 1.2, the ability to see *all* of the relationships between the structural relationships in the system, was overwhelming.

In evaluating this prototype we found that:

- Industrial developers were able to successfully plan pragmatic reuse tasks on their own tasks using Gilligan.
- Industrial developers felt they would be better able to understand larger pragmatic reuse tasks using Gilligan than were they to undertake the same task without our tool support.

- While the graph-based metaphor for planning pragmatic reuse tasks was effective, developers felt that it provided them with more information than they needed to be successful. Based on this feedback the planning interface was rebuilt completely for the next prototype.

Through conversations with our industrial participants, we also learned that they felt that while planning a pragmatic reuse task may be helpful, Gilligan would be much more valuable for them if it assisted them in performing their pragmatic reuse task, in addition to planning it. As an aside, we discovered that running industrial developers through controlled experiments may be more effective; several of our industrial developers had to receive managerial approval (in one case to the vice-presidential level) and legal approval to release their case study data to us. We could avoid these problems and still continue to work with industrial developers in the future, by performing controlled experiments whereby we control the setting by providing the tasks. In these experiments we would instead be measuring industrial developers performance on our set tasks and relying on their qualitative feedback to relate our chosen tasks to their industrial experiences.

Chapter 7

Tree-based pragmatic reuse planning

While evaluating our first Gilligan prototype with industrial developers, we learned that these developers did not think about the structural dependencies of a system in the way that we expected while investigating a pragmatic reuse task. The difference can be enumerated into four main observations:

- In order to scale to larger tasks, developers are willing to forego having all of the details about the structural relationships in the system available to them in the abstraction, if it means the task is more tractable and less overwhelming. If they feel like they need further information they are comfortable going to the source code to get it.
- While planning a pragmatic reuse task, the developer is mainly interested in *outgoing* dependencies. *What does this call? What does it reference? What classes and interfaces does it inherit from?* This focus arises from the fact that they want to learn how the code they are investigating depends on the system, not how the system depends on it.
- There are two main groups of dependencies developers are interested in while investigating any structural element, its direct dependencies (what does it call, reference, and inherit from) and its indirect dependencies (what is the transitive closure of all its dependencies).
- Developers think about the dependencies of a structural element more like a tree than a graph.

The results of the prototype and evaluation discussed in this chapter have been previously published [Holmes and Walker, 2007b].

7.1 Design goals

Based on these observations, we investigated the pragmatic reuse planning problem from a different direction: what information does the developer *require* while creating a pragmatic reuse plan? Using this information, we changed our goals from the first prototype (Section 7.1) and added some new goals to follow while designing our second prototype.

DG 1.1: *Provide an abstract representation of the structural elements and relationships being investigated.* Unchanged.

~~**DG 1.2:** *Visualize all the structural elements and their relationships.*~~ Developers found that considering all the relationships had scalability issues. This goal has been replaced by DG 2.1.

DG 1.3: *Promote easy navigation between structural elements.* Developers had some difficulty navigating using the graphical view provided by the first prototype. The developers often referred back to the tree view to locate the node they wanted there so it would be highlighted in the graph for them. This goal has been augmented by DG 2.3.

DG 1.4: *Provide a high-level overview of the reuse task.* Unchanged.

DG 1.5: *Explicitly record decisions about structural elements.* Unchanged.

DG 1.6: *Make those structural elements that have been triaged easily differentiable from those that have not.* Unchanged.

DG 1.7: *Encourage making performing pragmatic reuse tasks more systematic.* Developers found it difficult to progress through the graph systematically as the nodes were not structured in a way that promoted sequential navigation. While we arranged the nodes in this way to facilitate understanding, this decision interfered with working through the nodes one at a time. This goal has been augmented by DG 2.2 and DG 2.3.

DG 1.8: *Conscientiously avoid work that does not directly aid in the completion of the reuse task.* Developers made specific comments about enacting pragmatic reuse plans. This goal was investigated in more detail with the third prototype (Chapter 8).

DG 2.1: *Focus on outgoing dependencies.* Based on developer feedback, our new prototype should focus on delivering outgoing dependencies to developers, rather than trying to describe both incoming and outgoing dependencies.

DG 2.2: *Make it easier to identify both the direct dependencies and all the indirect dependencies.* Developers consider these two categories separately. While the direct dependencies give the developer a feel for what the system is doing, the indirect dependencies are more indicative of the cost of reusing a dependency; however, it is not possible to understand the indirect dependencies of any structural element without investigating many source files.

DG 2.3: *Use standard UI widgets that developers are familiar with.* Based on feedback from our first prototype we decided to try to use standard UI widgets such as lists, trees, and

editors in the second version of Gilligan, rather than focus on a graphical approach. This goal was added to increase keyboard accessibility and to reduce the training required for a developer to learn how to use Gilligan effectively.

7.2 Second Gilligan prototype

Similar to our first prototype, the main objective of our second Gilligan prototype was to help developers plan pragmatic reuse tasks. The user interface of this prototype was completely rewritten taking into account the feedback we received from our industrial participants after they used the first prototype of the tool. This prototype aims to better support developers performing these tasks by providing only the subset of features they need [Reiss, 2005] [**DG 2.1, 2.2**]; if they require more information than the tool provides they can use the source code directly to gain a greater understanding. Our primary concern with this prototype was to reduce the amount of work required of the developer while investigating the structural relationships within the system. By leveraging standard UI widgets we aim to make Gilligan easier to learn and make the navigation aspect quicker to use [**DG 2.3**].

At the conclusion of the industrial case studies (Section 6.3) we solicited feedback from the industrial developers involved in the evaluation and performed some participatory design sessions with them including brainstorming about solutions to the shortcomings identified from using the first prototype to plan industrial pragmatic reuse tasks. These sessions included creating lists of information needs, creating paper mock-ups, white-board designs, and use cases. By taking a more user-centred approach, the second prototype aimed to better address the needs of industrial developers when creating pragmatic reuse plans.

As with the first prototype, starting a pragmatic reuse task can be done with only a few clicks of the mouse. Figure 7.1 contains a screen capture of the start task dialog; here the developer has selected a source and target project and provided a short textual description of the reuse task. Once the dialog closes, the projects are statically analyzed.

Gilligan requires a starting point to be selected before the developer can navigate through the structural dependencies in the system. Figure 7.2 contains a screen capture of what this process entails; in this case the developer has searched for `*drawChart*`. Only elements from the source system are used to populate this dialog.

The Gilligan planning UI with the initial `drawChart` method is shown in Figure 7.3. The Gilligan UI is made up of four main panels. The three panels along the top are structural exploration views; this is where the developer can explore the abstraction of the source code they are investigating for reuse. The bottom panel contains the source code view. The developer is free to resize the views as it suits them; if they are not using the source code view they can make the top views taller (or conversely they can shrink them if they are using the source code extensively).

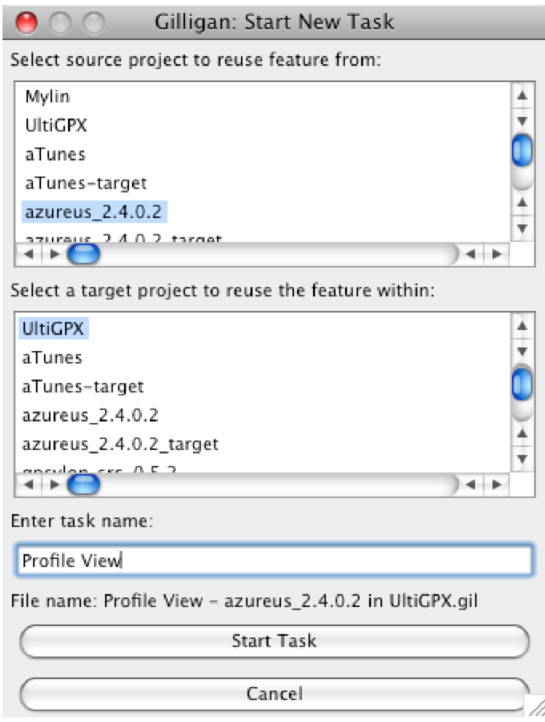


Figure 7.1: The initial dialog for starting a pragmatic reuse task with Gilligan. The developer needs only to select the source and target projects; the textual description is optional.

The leftmost panel in Figure 7.2 is used to track any structural element that the developer has tagged with a decision. If at any point they tag a structural element in any view, the element is automatically added to the leftmost panel [**DG 1.4, 1.6**]. This panel acts as a placeholder for significant structural elements for the developer. If the developer wishes they can also manually add structural elements that have not been triaged to this view by pressing the small 'plus' icon above the view.

The middle panel in Figure 7.2 displays only the direct dependencies of any element selected in the leftmost view. The rightmost panel in Figure 7.2 displays only the transitive closure of any element selected in the middle view. These two panels make it easy for a developer to quickly see both the direct and indirect dependencies quickly and concurrently [**DG 2.2**].

Each of the structural panels have several common elements (Figure 7.4 contains a screen capture of one of these views). Each panel has a text filter box along its top margin. This filter can be used to find a specific element (or group of elements) in a view using either plain text or regular expressions; changing the filter does not affect the selection allowing the developer to filter without modifying the state of the other panels. Each panel consists of four columns. The description (leftmost) column contains a tree that contains the name of the structural element, shows its containment hierarchy, has an icon showing the kind of element it is, and gives a visual cue showing if the developer has selected that element yet or not. If a developer has selected an element, its text is shown in black; if

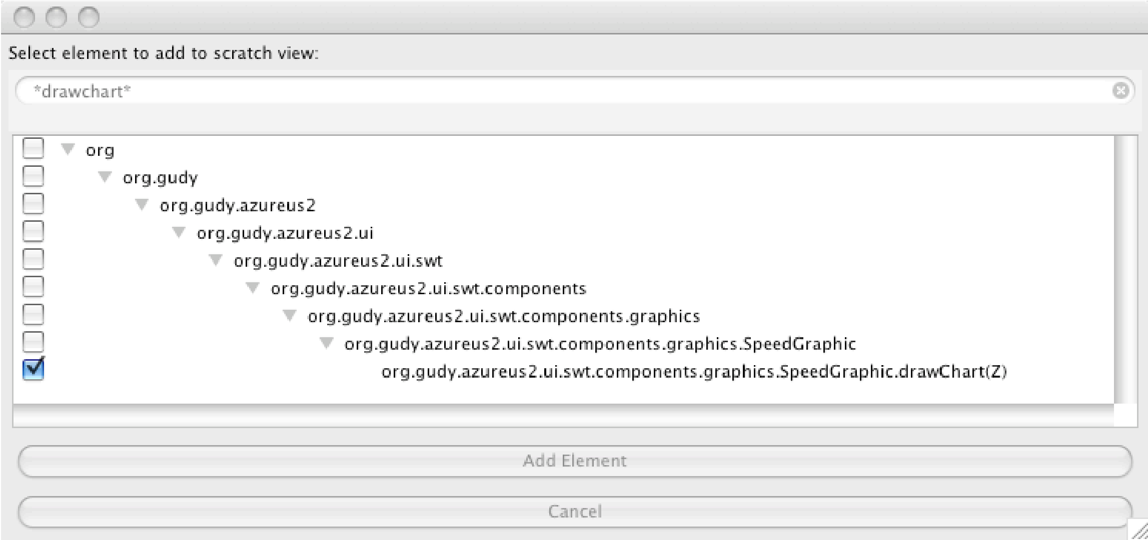


Figure 7.2: Gilligan dialog for selecting the starting point for investigation; the developer can select as many structural elements as they wish. Additionally, the developer can reactivate this dialog at any time to manually add any structural element into their reuse plan.

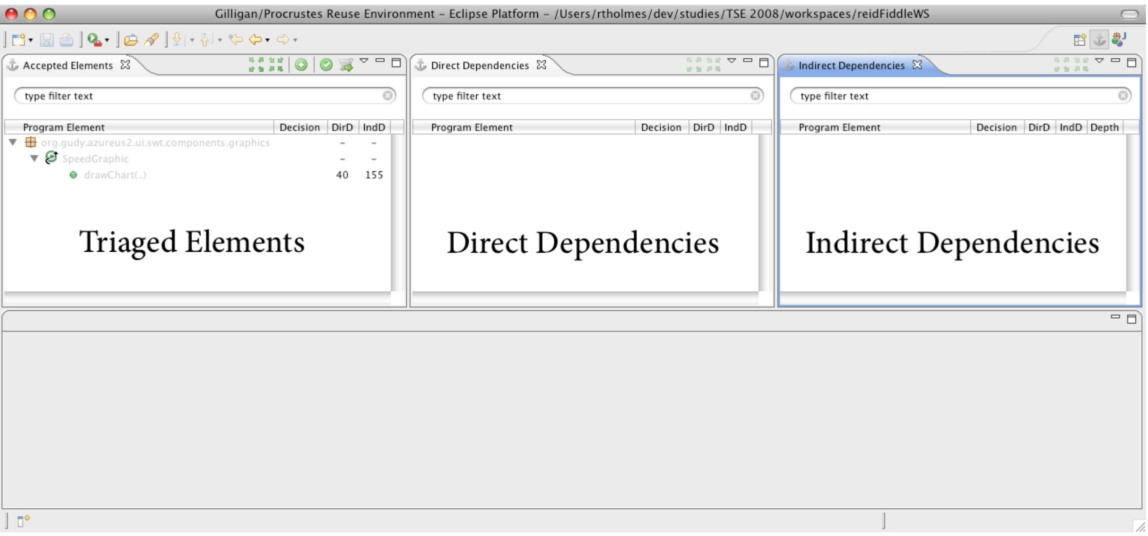


Figure 7.3: Initial view presented by the second Gilligan prototype.

they have not yet visited an element explicitly, its text is shown in grey. The decision (second) column displays any decision the developer made about a structural element [DG 1.5]. Unlike the first prototype, this decoration was moved to its own column to improve readability. The direct (third from the left) dependency count column gives the developer a preview of the number of dependencies an element has without having to select it. The indirect (rightmost) dependency count column informs the developer about the number of structural elements in the transitive closure for that element. For

example, in Figure 7.4 `AEMonitor.enter()` has 12 direct dependencies and 113 indirect dependencies. The direct and indirect dependency counts have been added to give the developer additional information about the potential "cost" of a dependency without forcing them to navigate through its dependencies to discover this information manually [**DG 2.1**].

In the first prototype developers frequently asked to be able to select multiple elements simultaneously so they could make many decisions at once or so they could look at many sets of dependencies concurrently. This second prototype enables any arbitrary set of dependencies to be selected in any view; however, the view they are selected in has direct bearing on what resultant relationships Gilligan populates the views with.

Figure 7.5 contains a series of screen captures showing how selections work in Gilligan (this example is taken from the scenario from Chapter 2). Once the developer starts the reuse task (Figure 7.1) and selects `drawChart(..)` as the initial starting point (Figure 7.2), Gilligan provides the initial view shown in Figure 7.5(a).

Any element selected in the leftmost view will have its direct dependencies shown in the middle panel. Figure 7.5(b) demonstrates what happens once the developer selects `drawChart(..)` from the leftmost panel. If multiple elements are selected, the union of their direct dependencies is shown in the central panel.

The transitive closure of any element selected in the central view will be shown in the rightmost view. Figure 7.5(c) demonstrates the effect of selecting all of the elements in the central panel; in this case the transitive closure of the dependencies for `drawChart(..)` are shown. Selections in the rightmost view have no effect because by definition the transitive closure of any element in the rightmost panel is already shown. If a developer wants to quickly investigate a structural element in the leftmost panel, they can simply double click on it: its direct dependencies will be shown in the middle view, they will be selected, and the transitive closure will appear in the rightmost view [**DG 2.2**].

For any selected element, Gilligan only shows the structural dependencies and their containing element(s). The dependencies that are show are only calls, references, has type, and inherits from [**DG 2.1**]. Some structural information is not shown, e.g., in Figure 7.5(b) the developer has selected the `drawChart(..)` method. In the direct dependency view, the `AEMonitor` class is listed containing two methods (`enter()` and `exit()`) even though this class declares 6 methods and 4 fields. These other structural elements are not listed because `drawChart(..)` is not directly dependent upon them directly. As structural elements are sometimes hidden (e.g., those methods and fields not shown for `AEMonitor`), developers have the options to request all of the contained structural elements for any element; this action can be accessed via the context menu.

Three types of additional information are presented to the developers through icon overlays, as is common practice in the Eclipse IDE. Table 7.1 shows the three kinds of icon overlays; these overlays can occur in any combination but are only shown (and are only relevant) for classes and interfaces.

Icon Overlay	Description
	Contains elements that are not shown
	Has at least one superclass or superinterface
	Has at least one subclass or subinterface

Table 7.1: Icon overlays used by Gilligan to show additional information.

As developers are used to working with source code, we augmented the source code editor (rather than a source code view as in the first prototype) to visually display the developers decisions to them by colouring the source code directly [**DG 2.3**]. The source code is displayed using the same colours as the decisions in the reuse plan. Two screen captures of the source code mark up are provided in Figure 7.6.

7.2.1 Application to the motivational scenario

Applying the second Gilligan prototype to the motivational scenario described in Chapter 2 was in effect similar to applying the first prototype. The same decisions were needed; however, using this prototype, the developer had to click on significantly fewer elements. They could also clearly see that `AEMonitor` and `COConfigurationManager` would involve a large number of dependencies by looking at the direct and indirect dependency counts in the structural views. The developer would also have to spend less time navigating through the graphical view because they can just click and filter on the elements they are interested in in the structural views. Figure 7.7 contains a screen capture showing some of the interesting elements for this reuse task and how they would look with the new prototype; this can be compared to Figure 6.3.

7.3 Evaluation

Our first prototype investigated the questions "can industrial developers create pragmatic reuse plans using Gilligan?" and "do these developers perceive any benefit planning their reuse tasks with Gilligan?". We wanted to investigate this prototype in a different way. The second prototype focused on improving Gilligan's ability to deliver the information that developers need, and to do this with less work on their part. Our research question for this evaluation was:

- Can developers more effectively locate structural dependencies using Gilligan compared to their normal practice?

To evaluate this research question, we conducted a semi-controlled experiment using six participants, each of whom performed four tasks. Two tasks were undertaken using our approach, while the other two tasks were performed using standard IDE tools ("manually"). The participants were asked to identify the transitive dependencies for a fragment of source code. This task was selected

because it is relevant to a developer performing a pragmatic reuse task who is asking, *If I reuse this dependency, what else will I have to bring with it?* Participants were given an unlimited amount of time to complete each task and were passively monitored during this time. Each participant was a software engineering graduate student who was an experienced Eclipse IDE user and had developed software on a regular basis.

The tasks undertaken were selected by identifying four Java-based open source projects and randomly searching for methods that had between 25 and 100 transitive dependencies. This problem size was chosen to enable the tasks to be completed in 10 to 20 minutes so each subject could perform 4 tasks within a reasonable amount of time. The subjects performing the manual treatments were provided with a simple tool to record the dependencies that they deemed relevant to the task.

Subjects (S1 through S6) were assigned to treatment groups (G1, G2) in a randomized fashion. We chose to keep the task order fixed (T1 through T4) to keep any learning effects consistent between subjects. The subjects in G1 performed T1 and T2 with Gilligan and T3 and T4 manually, while the subjects in G2 reversed this treatment. Additionally, to decrease learning effects, we chose each task from different systems.

Before the participants took part in the study, they were each given a written description for each task as well as a training task they could perform using both Gilligan and the manual approach. The subjects had as long as they wanted to investigate the tools on the training task before starting the four experimental tasks.

We recorded how long each task took and which dependencies the subjects identified for the source fragment they were investigating. We compared these results to solutions we derived by carefully completing the task several times; we use this experience to claim the number of correct, missing, and incorrect nodes for each trial. Using these numbers we are able to report precision (ratio of relevant nodes identified to relevant and irrelevant nodes identified) and recall (ratio of relevant nodes identified to number of relevant nodes in solution) of the technique.

We employed the Concern Mapper Eclipse plug-in[1] to permit the participants to record dependencies easily during the manual case. The Concern Mapper plug-in simply records a user-selected list of structural entities within the IDE. As developers encountered entities upon which the code fragment of interest had direct or indirect dependencies, they were told to add them to the plug-in's view. Concern Mapper also served as an augmentation of the standard IDE toolset, as the developer could use it as an index to click on any entity to switch to it. This was more manageable than working through the list of all the elements in the system (or using the open editors).

[1]http://www.cs.mcgill.ca/ martin/cm/ v1.3.1

Task	Classes	Fields	Methods	Total
Ganymede	4	9	17	30
HttpClient	7	22	31	60
GanttManager	12	10	28	50
Jajuk	15	47	31	93

Table 7.2: Size of the correct solutions.

7.3.1 Experimental tasks

Next we describe each of tasks the subjects were to perform. The number of dependencies in the solution for each task is shown in Table 7.2. The correct solutions for each of the tasks are included in Appendix D.

Task 1: Ganymede. The first task involved investigating the dependencies for starting a Log4J socket server inside Ganymede,[2] an Eclipse-based viewer for receiving Log4J notifications over sockets. Ganymede consists of 34 classes and 2,234 lines of code. The subjects were tasked with finding all of the transitive dependencies for `Log4JServer.startListener()` while excluding any dependencies from the Java standard libraries or from the `org.eclipse.*` packages.

Task 2: HttpClient. For the second task the subjects had to identify the dependencies for parsing cookies in HttpClient.[3] HttpClient is a library that provides a level of abstraction above the `java.net` layer, implementing a full HTTP stack for developers to use in their own systems. This project contains 165 classes and 15,970 lines of code. The subjects started in `CookieSpecBase.parse(..)` and excluded any Java standard library classes.

Task 3: GanttManager. The third task involved locating the dependencies for adding a delay to a Gantt activity in GanttManager.[4] The Gantt project provides a tool to help users create project schedules consisting of tasks and resources assigned to complete these tasks. The Gantt project consists of 555 classes and 43,247 lines of code. Each participant started in `ConstraintImpl.addDelay(..)`. Java standard library dependencies were excluded.

Task 4: Jajuk. The final task looked at adding an item to a playlist inside the Jajuk media manager.[5] Jajuk is a comprehensive media manager and player written in Java; it consists of 227 classes and 30,679 lines of code. The participants were asked to ignore dependencies on Java standard libraries and on `org.apache.logging.*` packages. Each participant started their investigation in `History.addItem(..)`.

[2]http://ganymede.sf.net (v0.9.3.1)
[3]http://jakarta.apache.org/httpclient (v3.0_rc1)
[4]http://ganttproject.sf.net (v2.0.4)
[5]http://jajuk.info (v1.3.0)

7.3.2 Results

After analyzing the trace data that we had collected while the subjects performed each task, we were able to forensically determine if and when they deviated from the correct solution.

Task 1: Ganymede. S1, S2, and S3 used Gilligan for this task, finding all the correct dependencies in an average of 5 minutes and 20 seconds. Performing the task manually, participants S4, S5, and S6 had a recall of 0.77 and took an average of 13 minutes 40 seconds. The recall of S4–S6 suffered because each of them failed to adequately explore the dependencies of `GanymedeUtilities.initActions()`. This one method call was obscured as a condition in an `if` statement but was the root of a dependency chain containing over 18 other elements. Although S4–S6 identified many of these dependencies via other paths, they ended up neglecting to identify an average of 7 structural dependencies from the solution, each of which the subjects using Gilligan had found.

Task 2: HttpClient. The participants manually performing this task identified far fewer correct dependencies than the participants supported by Gilligan (46% vs. 99%). Again, participants manually performing the task missed a large number of dependencies by failing to follow a single path through `Date.parseDate(..)`; this path led to 13 dependencies on `ParameterParser` that each of these subjects failed to detect.

Task 3: GanttManager. In this task, both treatment groups failed to detect a large number of the structural dependencies. Participants in the manual treatment group had an average recall of only 39% over an average of 17 minutes of investigation while the Gilligan-supported subjects had an average recall of 50% over 8 minutes and 20 seconds. While observing the participants performing this task, 5 of the 6 failed to notice that half of the structural dependencies for this task were performed by subtypes that needed to be investigated indirectly. Several of the structural dependencies led the participants to interfaces and abstract classes; only one of the participants investigated the subtypes of these interfaces. In each case only one subtype existed so it was easy to determine which type the code was actually dependent on.

Task 4: Jajuk. This task had the greatest discrepancy between the Gilligan-supported subjects and the manual participants. The manual-treatment subjects had an average recall of 19% in 9 minutes and 40 seconds. The participants using Gilligan had an average recall of 98%, spending 11 minutes and 20 seconds to identify these dependencies. Although the manual-treatment subjects performed the task on average slightly faster than the Gilligan-supported subjects, they performed significantly worse in locating each of the correct dependencies. During this task we observed the participants revisiting the same node much more than in other tasks. The manual-treatment participants in particular would visit the same method several times trying to remember if they had been there before. The manual participants each missed a branch containing 77 dependencies by not following a single path into `FileManager.getInstance()`.

Precision and recall. We compared the dependencies identified by the subjects to the solution for each task. Recall rates, that is, the proportion of correct dependencies identified by each subject for each task are shown in Figure 7.8. From this graph we can see that subjects manually carrying out the tasks generally did worse than those supported by Gilligan. Neither approach yielded high recall for T3. Over all the tasks, the average recall for the Gilligan-supported subjects was 87% while the manual-treatment participants achieved 45%.

We also calculated precision; that is the proportion of invalid nodes to valid nodes. The manual-treatment participants achieved an average precision of 89% while the Gilligan-supported subjects achieved an average of 98%. Precision was fairly consistent between each of the tasks, so we do not present it in a graph.

Figure 7.9 compares the average time taken per-task against the average recall. In this chart better solutions would appear at the bottom right, while worse solutions would appear at the top left. In general from this graph we can see that the manual tasks took longer and had lower recall than the tasks for which the subjects had access Gilligan. The subjects took an average of 9 minutes per task using Gilligan and 14 minutes performing the tasks manually. Their average recall using Gilligan was 87% compared to 45% in the manual case.

7.3.3 Observations

Observing each of the participants while they performed the tasks yielded several insights. During each manual task, the participants had difficulty determining if dependencies were part of the restricted set of dependencies they did not have to consider. The participants would hover over the dependency to see its fully-qualified type or they would navigate to the dependency and scroll to the top to see its package signature. For method bodies that had a large number of dependencies that met the restrictions, the subjects often missed important dependencies as they were obscured by unimportant ones; this was the case that led to the missed path in T2.

While using Gilligan, the participants would scroll though the transitive dependency list to check for any nodes that had not been annotated as a final step. While this was effective for T1, T2, and T4, it led to errors in T3 as the subjects needed to interact more with the tool to request the subtypes of the interfaces they had located (possibly pointing to an inadequacy in the tool). This shortcoming was partially addressed by the validation views provided by the third prototype (Chapter 8).

After the participants had completed all four treatments, we performed an exit interview to see how they perceived their performance for each treatment. The participants were quite positive about their performance: on a Likert scale from 1 (very poor) to 7 (very high) they rated their confidence in their solution at an average of 6.7 for the tool-supported cases and 5 for the manual cases. The most common complaint about the manual approach was that "since you can't remember where you've been you often end up doing things over" (S4); these sentiments were echoed in similar comments by S2, S5, and S6. S1 also stated that dependencies could sometimes be "obfuscated by the source

code" and that "scanning through code your brain sees chunks and if you misinterpret some part you'll just skip over [it]". This was borne out in their performance of the manual tasks.

7.4 Discussion

This evaluation was interesting because it pitted our tool, that developers had received less than 10 minutes worth of training for, against all their experience using an IDE to locate dependencies. While we had hoped developers would be more effective (or at least faster) using Gilligan, we were pleased to find that both seemed to be true. One surprising result of this evaluation was how much trouble developers had trying to identify structural dependencies in the manual case; we were surprised to find that in all the 12 manual cases, none of the developers managed to find all of the relevant dependencies (using Gilligan, developers found all of the correct dependencies in 7 of 12 cases).

While Gilligan performed well during the evaluation, in the third task the subjects failed to locate many of the relevant dependencies. This shortcoming occurred because, by default, the tool is designed to show dependencies from method calls and field references. In this task, many of the dependencies originated in subtypes in the inheritance hierarchy. While visual cues for this information were provided by the tool, we believe that they were not effective because the developers trusted its default output. Gilligan will have to be modified to provide more inheritance-aware information by default in the future (this is discussed in future work (Section 10.4)).

The ease with which the developers learned and used Gilligan reinforced our belief that DG 2.3 was the right decision; the developers were able to quickly and easily adapt to the UI widgets with which they were accustomed. The success the developers had using Gilligan was a positive sign that the decisions we made to support DG 2.2 were effective although further evaluation on real reuse tasks would be needed to verify this. While this prototype did focus on outgoing dependencies as required by DG 2.1, this evaluation did not address whether this was the right decision or not.

Switching to the tree-based method displaying and navigating dependencies caused some interesting functionality to be lost that was present in the first prototype. With our tree representation it is not easy to see how 'important' any structural element is, whereas with the graphical view a developer could look at any node in the graph and enumerate its in-edges. Also, it is not easy to get a feel for the size of the reused feature in relationship to the transitive closure that feature could have and the size of the whole system. While this was not easy with the graphical view, it could be done if the developer really wanted to know. Ultimately, maintaining some of the overview-perspective aspects of the first prototype and providing a means to both navigate up and down dependencies could alleviate many of these shortcomings.

7.4.1 Limitations

We conducted our evaluation to gather initial evidence into the efficacy of our approach in order to judge whether further investment should be made in the tool. The small sample size and small scale

of our evaluation are obvious drawbacks. While our subjects were graduate students, they were all software engineers who actively write code, and many of them have industrial experience. While the tasks performed by the subjects were not very large, the average recall for the manual cases was quite low; using larger tasks would likely only exacerbate this result. Our semi-controlled experiment demonstrated, for the tasks and subjects we tested, that Gilligan increased recall and decreased the time required for subjects to identify the structural dependencies compared to manual techniques. Further evaluation involving industrial developers and tasks is needed to determine the generalizability of these results.

It would have been very interesting to compare developers searching for these dependencies using the first and second Gilligan prototypes. While anecdotally we believe that developers would have performed much better with the second prototype (at least in terms of time), this evaluation did not investigate that potentially interesting avenue.

7.5 Summary

Based on the feedback our industrial developers gave us in the first evaluation, we explored a different, tree-based metaphor for helping developers explore the structural dependencies within the source code with our second Gilligan prototype. This prototype added three new design goals, while eschewing one of the goals we had invalidated after evaluating our first prototype. Two of these design goals arose by carefully considering the information needs of a developer performing a pragmatic reuse task while the third involved using UI elements that developers are familiar with to reduce the learning curve for our tool.

Upon evaluating our second Gilligan prototype, we learned three primary lessons:

- Developers are surprisingly poor at locating all of the structural dependencies for a particular fragment of source code using standard development tools. This skill is important for assessing pragmatic reuse tasks because one missed dependency can be the difference between a task being easy or infeasible.

- A mechanism to alert developers to investigate dependencies they have omitted, or to let them know that they have investigated all the dependencies would be helpful.

- Developers could effectively navigate structural dependencies using the second Gilligan prototype.

This evaluation did not identify any glaring deficiencies with the exploration aspect of the Gilligan prototype, although we discovered we could improve the way that Gilligan represents type hierarchy information. Our final prototype instead investigated a piece of feedback we received from our industrial participants after they used the first prototype, that Gilligan should semi-automate the enactment of the pragmatic reuse plan.

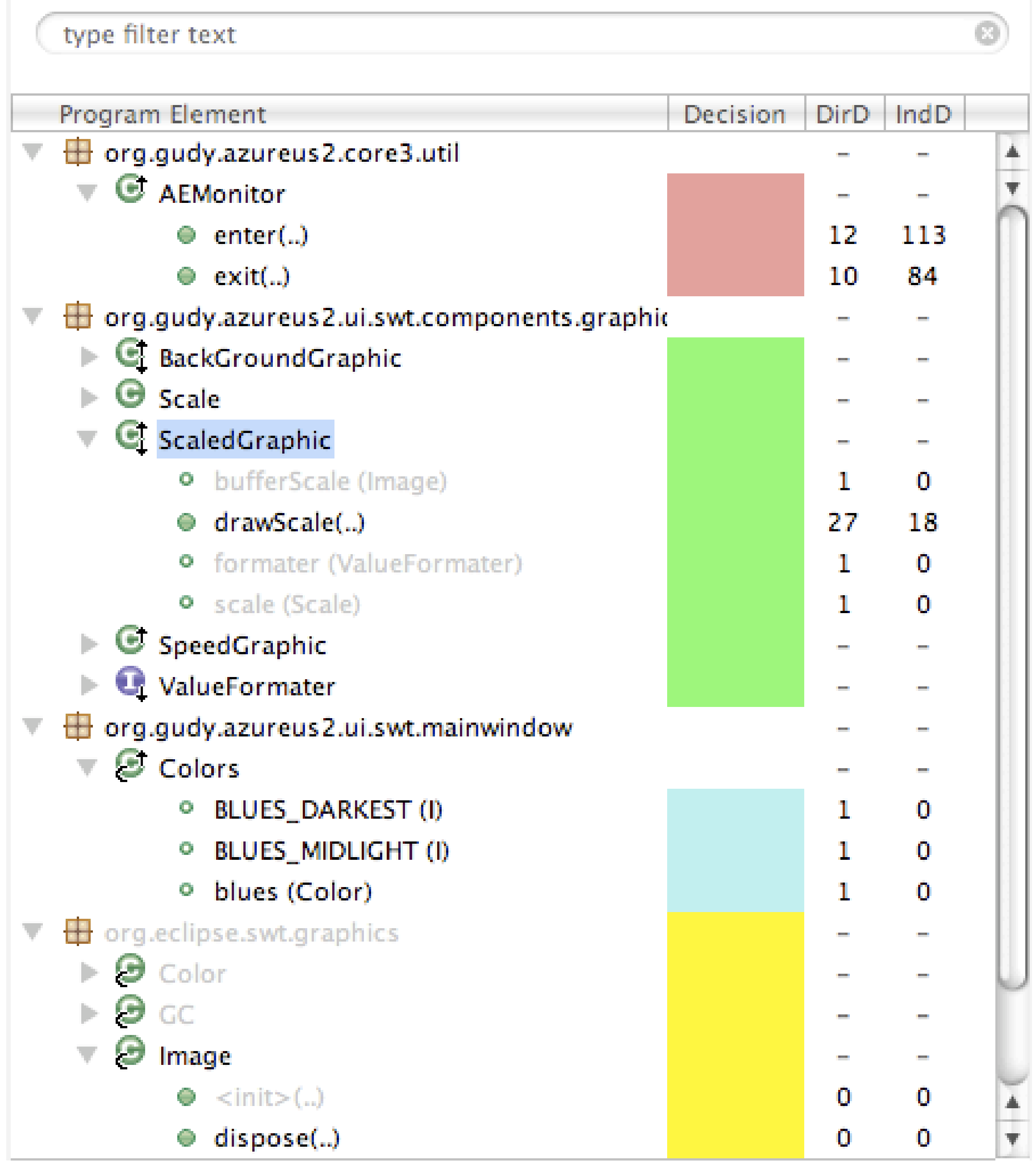
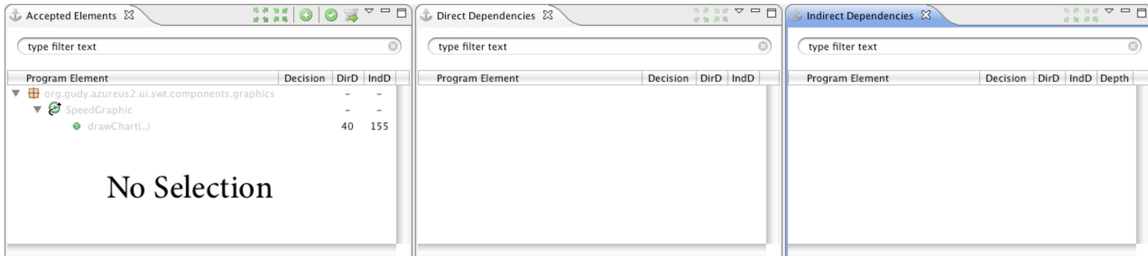
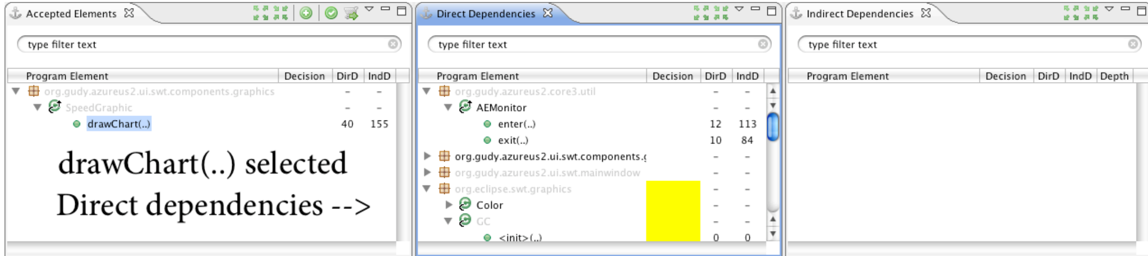


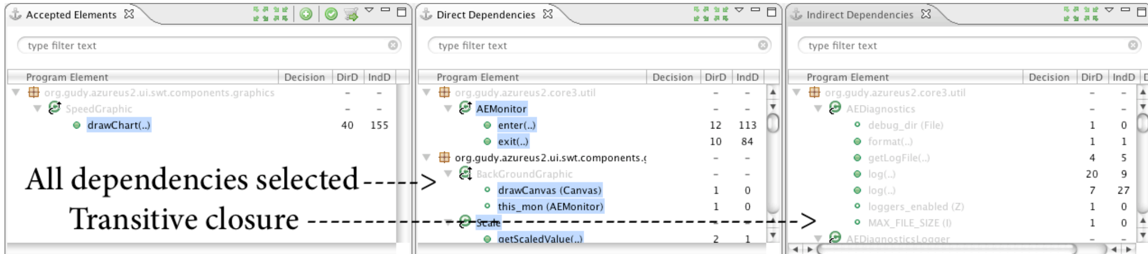
Figure 7.4: A Gilligan structural view showing each of the columns of information as well as every decision colour and many of the icon overlay combinations.



(a) No elements selected; `drawChart(..)` is light grey because it is yet to be selected. Even without selecting `drawChart(..)` the developer can see that it has 40 direct dependencies and 155 indirect dependencies.



(b) Once `drawChart(..)` has been selected the middle pane is populated with all of `drawChart(..)`'s direct dependencies. At a glance the developer can see that `AEMonitor.enter()` and `AEMonitor.exit()` are the root of many of `drawChart(..)`'s transitive dependencies.



(c) After the developer double clicks on `drawChart(..)` (or selects all of its direct dependencies in the middle view), the rightmost view is populated with the transitive closure of the dependencies required by `drawChart(..)`. If the developer wants to see the transitive closure of any specific direct dependency required by `drawChart(..)` they can select only those elements they are interested in in the middle view and the rightmost view will be updated.

Figure 7.5: Screen captures demonstrating Gilligan's selection behaviour.

```java
SpeedGraphic.java
protected void drawChart(boolean sizeChanged) {
  try{
     this_mon.enter();

     drawScale(sizeChanged);

     Rectangle bounds = drawCanvas.getClientArea();

     //If bufferedImage is not null, dispose it
     if(bufferImage != null && ! bufferImage.isDisposed())
       bufferImage.dispose();
```

(a) Initial view of the source code annotations for the `drawChart(..)` method. Common dependencies are already highlighted in yellow from the outset as Gilligan triages these automatically.

```java
SpeedGraphic.java
protected void drawChart(boolean sizeChanged) {
  try{
     this_mon.enter();

     drawScale(sizeChanged);

     Rectangle bounds = drawCanvas.getClientArea();

     //If bufferedImage is not null, dispose it
     if(bufferImage != null && ! bufferImage.isDisposed())
       bufferImage.dispose();
```

(b) Final view of the source code annotations for `drawChart(..)` after the developer has made several decisions. The `this_mon` field and call to `enter()` have been rejected by the developer. The `drawScale(..)` method has been accepted along with the `drawCanvas` and `bufferImage` fields.

Figure 7.6: Gilligan source code view before and after decisions were made.

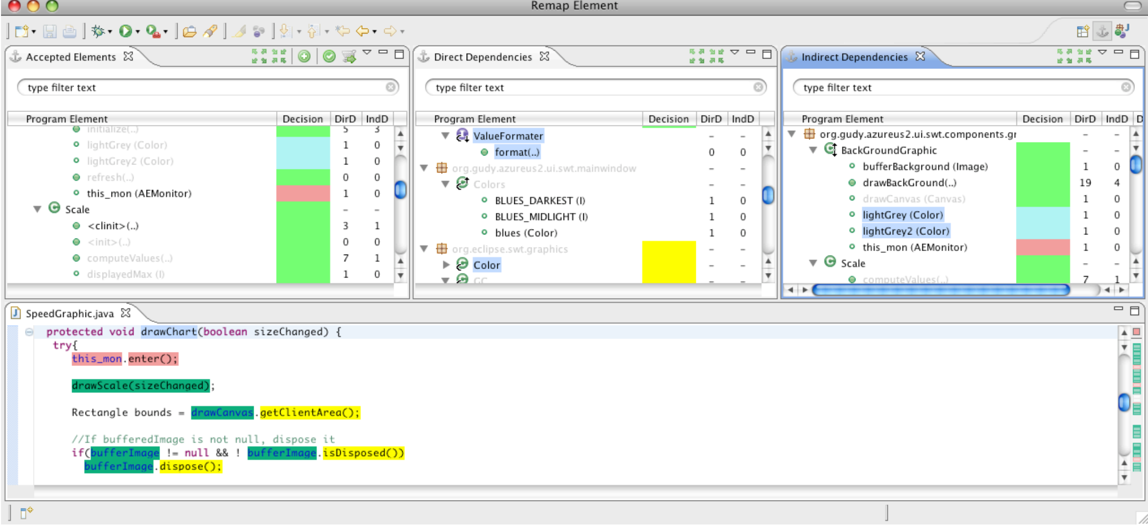


Figure 7.7: Screen capture while working on the motivational scenario.

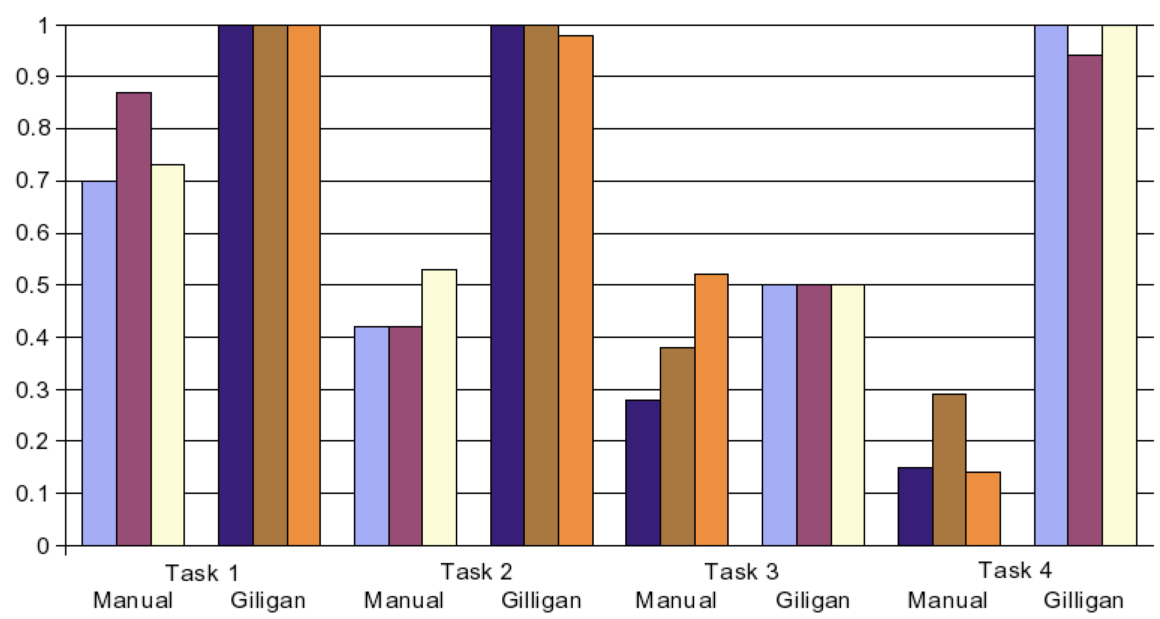


Figure 7.8: Recall for the four tasks.

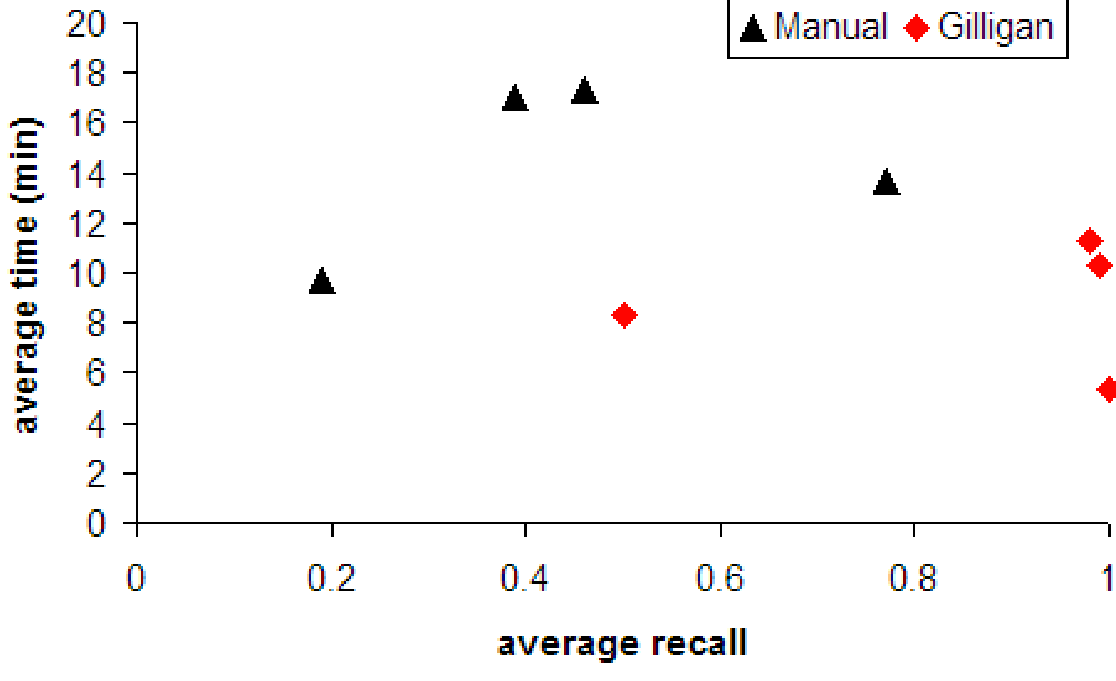


Figure 7.9: Average recall and time.

Chapter 8

Pragmatic reuse plan enactment

Our first two Gilligan prototypes concentrated on pragmatic reuse planning. Several major usability issues were identified when we evaluated the first prototype; after evaluating our second prototype, and watching developers work with it, we decided that we needed to improve several areas to help developers better *perform* pragmatic reuse tasks. Looking back at all the feedback we had received from our participants, and drawing on our own experience, we identified three key ways in which Gilligan could be improved:

- *Provide feedback about the completeness of the pragmatic reuse plan.* While evaluating the second prototype we would see developers identify all the correct dependencies and then continue investigating, just to be sure. Gilligan should direct developers to structural elements they have missed investigating, as well as let them know when they have investigated everything they should look at.

- *Make the reuse plan executable.* Pragmatic reuse plans are lightweight specifications describing not only what elements are involved in a pragmatic reuse task, but how they should be managed when the task is carried out. Gilligan should automatically extract the source code specified in the pragmatic reuse plan and integrate it into the target project, modifying it as necessary.

- *Developers are not omniscient.* Gilligan should help developers to perform pragmatic reuse tasks in a more iterative manner; that is, they should not have to create one perfect plan and execute that plan once, rather they should be able to make a decision and instantly see the effects of that decision in the source code.

The results of the prototype and evaluation discussed in this chapter have been previously published [Holmes and Walker, 2008].

8.1 Design goals

Based on these key themes, we identified three new goals that the third Gilligan prototype should meet. The new goals augment DG 1.8 and DG 2.1.

DG 1.1: *Provide an abstract representation of the structural elements and relationships being investigated.* Unchanged.

~~**DG 1.2:** *Visualize all the structural elements and their relationships.*~~ Superseded by DG 2.1.

DG 1.3: *Promote easy navigation between structural elements.* Augmented by DG 2.3.

DG 1.4: *Provide a high-level overview of the reuse task.* Unchanged.

DG 1.5: *Explicitly record decisions about structural elements.* Unchanged.

DG 1.6: *Make those structural elements that have been triaged easily differentiable from those that have not.* Unchanged.

DG 1.7: *Encourage making performing pragmatic reuse tasks more systematic.* Augmented by DG 2.2 and DG 2.3.

DG 1.8: *Conscientiously avoid work that does not directly aid in the completion of the reuse task.* This goal is heavily augmented by DG 3.2.

DG 2.1: *Focus on outgoing dependencies.* Augmented by DG 3.1.

DG 2.2: *Make it easier to identify both the direct dependencies and all the indirect dependencies.* Unchanged.

DG 2.3: *Use standard UI widgets that developers are familiar with.* Unchanged.

DG 3.1: *Support validating a pragmatic reuse plan.* Gilligan must be able to identify structural elements that the developer needs to consider before their plan can be considered 'complete'. It should also be able to indicate to the developer when the developer has investigated all relevant structural elements.

DG 3.2: *Semi-automatically enact a pragmatic reuse plan.* A pragmatic reuse plan contains much of the information needed to perform the pragmatic reuse task; a developer could either walk through the plan one element at a time performing each step in turn, or a tool could do all of that work for them. Gilligan should be able to take a reuse plan and the source and target system and enact the plan semi-automatically. Gilligan should also consider the decisions encoded in the reuse plan to manipulate the source code to reduce the number of simple compilation errors the developer must manually fix.

DG 3.3: *Support an iterative process for pragmatic reuse planning and enactment.* Pragmatic reuse tasks are not performed by creating a single, perfect, reuse plan followed by a set enactment process; the reality is much more iterative. Developers understand source code and are constantly thinking about how their decisions will be implemented in the code. Gilligan must support an iterative process for planning and performing pragmatic reuse tasks.

8.2 Third Gilligan prototype

Unlike our first and second prototypes, the main objective of our third and final Gilligan prototype was not to help developers plan pragmatic reuse tasks but to help them *perform* them. The third prototype focused on the executable aspect of pragmatic reuse plans: how much work can Gilligan automatically do for a developer given their pragmatic reuse plan? [**DG 3.2**] Is the specification sufficient to take care of many of the details for the developer or is it too lightweight? DG 3.2 primarily involves investigating these issues. DG 3.1 and DG 3.3 both arise from automated support. If it is easy to enact the plan, how can we support a plan being enacted iteratively as a developer converges on a solution? [**DG 3.3**] How can we direct their investigation when they have only a partially developed pragmatic reuse plan? [**DG 3.1**].

Figure 8.1 contains an updated version of the pragmatic reuse process (the previous version can be found in Figure 6.1). One major flaw in the previous version of the reuse process was that the back edge from `Modify source code` to `Investigate / triage` was far too late in the process. In the old process, the developer had to start performing the pragmatic reuse task before they could see the effects of any decision in the source code; once the developer got to this point they would be resistant to going back into the planning interface because if they changed their mind they would have to undo any edits they had made to the source code up to that point.

Where the first and second Gilligan prototypes originally aimed to capture the intent of the developer with respect to their plans for pragmatic reuse, the third prototype aims to bridge the gap between the intent of the plan and the realization of the task by semi-automatically enacting the pragmatic reuse plan for the developer. First, Gilligan migrates the source code fragments that the developer intends to reuse from the originating project into the target project (see Section 8.2.1) according to the reuse plan [**DG 3.2**]. Second, Gilligan mitigates the compilation problems in the reused code caused by dangling references that arose from removing the code from its originating environment (see Section 8.2.2) by semi-automatically manipulating the source code [**DG 3.2**]. The goal of the semi-automation is to reduce the effort needed by the developer to transition from planning a reuse task to its successful implementation. These steps also enable the developer to quickly iterate on their reuse plans allowing them to investigate alternative reuse scenarios in an effort to improve their reuse plan [**DG 3.3**]. By moving the ability to see, in the concrete realization of the reuse task (the source code), the effects of any decision up in the reuse process, the developer is encouraged to continue to use the planning interface and to test new hypotheses about their reuse task [**DG 1.8**].

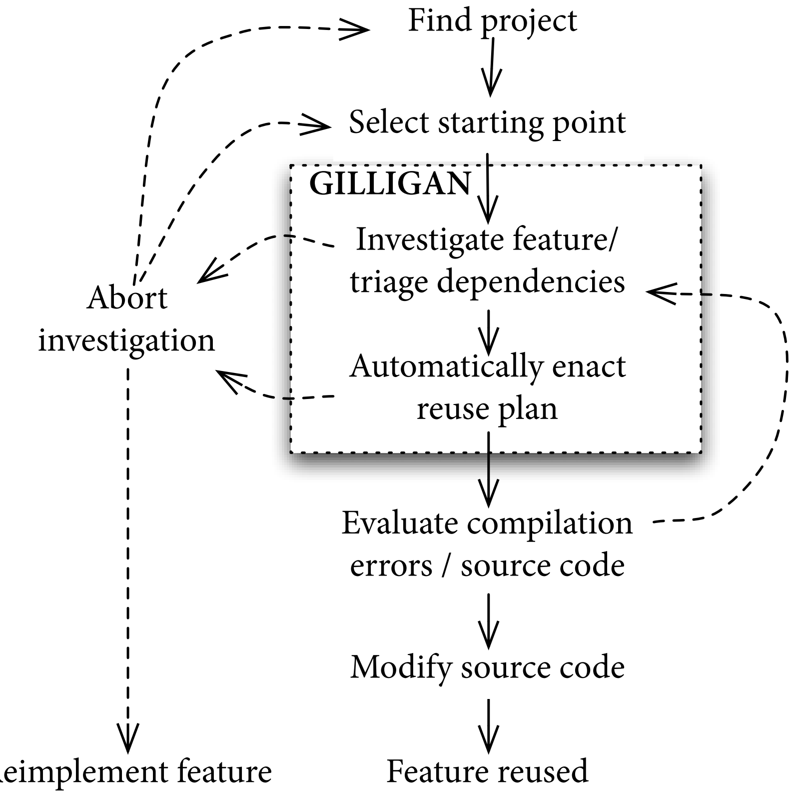


Figure 8.1: Gilligan's role in the pragmatic reuse process.

8.2.1 Extraction

The developer initiates the enactment of the pragmatic reuse plan by clicking the "enact plan" button. Gilligan then analyzes the plan to determine which source code entities should be reused. The tool locates all the source code corresponding to accepted nodes in the reuse plan. It then migrates the source code associated with these nodes into the target project [DG 3.2]. In order to keep the reused code separate from the rest of the source code in the target project, we create a source folder called `reusedCode` under which we keep all of the reused code, and augment the developer's build path to include this directory. This separation ensures that the reused code will not interfere with the developer's project and makes it easy for him to roll back from the reuse task by just deleting the generated folder. When code is extracted, the structure of its original package hierarchy is maintained for ease of comprehension.

We slightly restrict which nodes must be copied from one system to another. Any accepted class will be migrated; however, accepting any field or method implies that its declaring parent class is also accepted. This is because we cannot reuse code if we have nowhere for it to go. Only extracted fields are an exception to this condition. In addition, any non-annotated method or field of an accepted class will be reused but will not be modified in the integration phase. This restriction reduces the

complexity of the reuse plan and enables the reused source code to remain largely structurally intact relative to its originating system.

After Gilligan has copied the code into the target project, dependencies between the reused classes will remain valid as the package structure was maintained (these dependencies are represented by solid lines in Figure 2.5). Any dependencies to structural entities outside those being reused would normally cause compilation problems (these are represented by dotted and dashed lines in Figure 2.5); however, the integration portion of Gilligan resolves many of these (see Section 8.2.2). In Figure 2.5 `Graphic`, `ScaledGraphic`, `Scale`, and `ValueFormatter` have been extracted in their entirety. `SpeedGraphic` and `BackgroundGraphic` have been extracted but some of their methods and fields will be removed.

8.2.2 Integration

During this phase, the source code that has been migrated from the original system to the target system must be manipulated to resolve any compilation problems that have arisen. When source code is removed from the context for which it was written and placed into a foreign environment, many of its dependencies can be unfulfilled. The unfulfilled dependencies in the reused code are manifested as dangling references to types, methods, and fields that were not also reused (and do not exist in the target project).

Using the model backing the reuse plan, Gilligan can pre-compute each of the changes that the tool should perform to repair any dangling references. This model contains a static picture of every entity within the system that the reused code references. From this, Gilligan can predetermine which relationships are being deliberately severed or remapped in the reuse plan, and based on the annotations in the plan, it can determine how to manage these dependencies and automatically manipulate the source code to resolve many of the problems that arise from removing the code from its originating context [**DG 3.2**].

Gilligan proceeds in four major steps. First, it handles new code being added to the reused entities (Section 8.2.2.1). Second, it updates dangling references (Section 8.2.2.2). Third, it removes any unnecessary code (Section 8.2.2.3). And last, a simple finalization step takes place (Section 8.2.2.4).

8.2.2.1 Managing source code additions

The code addition step adds new code to those entities previously migrated to the target project in Section 8.2.1. There are two cases that must be handled: code injection, and field extraction. For code injection, any fragment provided by the developer is inserted into its target class (as specified in the reuse plan). For example, if the developer's plan specifies injecting the fragment "`Logger log = new Logger();`" into the `SpeedGraphic` class, the Abstract Syntax Tree (AST) for this class is loaded and the fragment is added to its beginning (e.g., as its first declaration). The burden is on the developer to ensure that the injected fragments make syntactic sense in the context of a class body

(e.g., injecting fields or methods is fine, but injecting an arbitrary fragment will likely result in an error). Code can only be injected into classes, not into methods.

For field extraction, any fields in the plan that have been marked for extraction are copied from the class in which they were declared into the target class specified in the pragmatic reuse plan. Moving the field only updates its declaration, not its references in the reused code (this happens in the next step). Again, the import statements are also updated to reflect this addition to the target class. For example, in the pragmatic reuse plan given in Figure 2.5, several fields in the `Color` class are extracted into the `SpeedGraphic` class. As classes other than `SpeedGraphic` use these fields, Gilligan also ensures that the `SpeedGraphic` class is public so these fields will be visible.

8.2.2.2 Managing dangling references

The management of dangling references is the most complex step in the integration phase. Gilligan proceeds through the pragmatic reuse plan minimizing the number of dangling references in the reused code. Two primary classifications of dangling references are managed: (1) references to fields, calls to methods, and references to supertypes that were rejected in the reuse plan; and (2) calls to methods and referenced fields that have been injected, extracted, or remapped in the reuse plan.

Gilligan starts by managing references to fields, methods, and supertypes that were not reused. Gilligan searches each accepted node for dependencies on other nodes that have been rejected. If a dependency to a field or method is found, it is managed: Gilligan comments-out the code corresponding to the dependency within the accepted code. (Currently, the alternative of "stubbing-out" rejected dependencies—by calling a dummy method or returning a nonce value—is not supported by Gilligan.) Gilligan comments out changes made to the bodies of methods due to calls and references being rejected, rather than remove them completely, as their details could still be informative to the developer. These comments are accompanied with a note to indicate that the change was made by Gilligan. This also allows the developer to easily locate each change to the source made by Gilligan using traditional search tools. For the sake of simplicity,[1] Gilligan rejects field references and method calls only at the statement level. For example, in Figure 2.5, the field `BackgroundGraphic.this_mon` is rejected; references to this field must be eliminated. `SpeedGraphic` references this field repeatedly by calling a method on it (e.g., `this_mon.enter()`). Because Gilligan eliminates references at the statement level, the whole statement is commented-out rather than just the fragment mentioning `this_mon`. In this case, this treatment suffices as the call to `enter()` is also rejected and would have had to be eliminated in a later step. As rejected elements take preference over accepted ones, if `enter()` had been accepted in this example but `this_mon` were rejected, the statement would still be commented-out, although the `enter()` method itself would still be reused.

More complicated examples can lead to problems. If, for instance, `getActionBars()` were rejected in the code snippet in Figure 8.2, Gilligan would simply comment out the whole statement,

[1]The Eclipse tooling for manipulating abstract syntax trees (ASTs) restricts where comments can be inserted.

Original code:

```
IStatusLineManager ism = getViewSite().getActionBars()
    .getStatusLineManager();
```

After rejecting the call to `getActionBars()`*:*

```
/** GILLIGAN: Call rejected in reuse plan.
IStatusLineManager ism = getViewSite().getActionBars()
  .getStatusLineManager();
*/
```

Figure 8.2: Example of how Gilligan rejects structural elements.

rather than commenting out the single method call. In this case, later statements that reference `ism` will also have to be removed (manually) as commenting out this line removes the variable declaration from the code.

If the pragmatic reuse plan has reused a class but not some subset of its supertypes, the tool must then remove these references. This often occurs as developers trim the functionality they are interested in from an inheritance hierarchy. Any number of supertypes can be removed. For example, in Figure 2.5, the class declaration `class SpeedGraphic extends ScaledGraphic implements ParameterListener` is updated because `ParameterListener` is rejected. Gilligan updates the declaration to be `class SpeedGraphic extends ScaledGraphic`. If the subclass were dependent on a method within a rejected supertype, this would be shown as a dependency between a method in the subtype and a method on the supertype during the planning process. This dependency would have been resolved at the beginning of this step.

Finally, any accepted node with a structural dependency that has been remapped is handled. These cases are simpler than in the rejected-node case as code does not disappear; it is simply redirected. This step handles 5 cases: calls to injected and remapped methods and references to injected, extracted, and remapped fields. For example, in Figure 2.5, `Colors.black` is extracted to the `SpeedGraphic` class; any reference to `Colors.black` within an accepted node will be remapped to `SpeedGraphic.black`. References in `BackgroundGraphic.drawBackground()` and `ScaledGraphic.drawScale()` must be updated to the extracted location (`SpeedGraphic.black`). If the reuse plan remapped logging functionality from `SWT.error(...)` to a method on the injected field `log`, such as `log.error(...)`, any reference to `SWT.error(...)` would be updated to `log.error(...)`.

8.2.2.3 Managing unnecessary code

This step removes extraneous reused code from the target system, that is, methods and fields marked as rejected in the reuse plan that are declared within accepted classes and interfaces. In the previous steps, any changes Gilligan made to a source file either added code, updated existing code, or removed sections of code by commenting them out. In this step, we found that removing rejected methods and fields by commenting them out made the source files seem cluttered by the number and size of the multi-line comments spread throughout the file. To address this, rejected fields and methods are completely removed from the source code by removing them from the AST of their containing class. Gilligan does not apply any modifications to rejected entities; it just eliminates them from the source. For example, `SpeedGraphic.getInstance()` makes reference to methods in the rejected type `COConfigurationManager`. As `getInstance()` is rejected, Gilligan did not resolve its dangling references in step two. The tool only needs to remove methods and fields that are children of classes that have been accepted, i.e., if a type is completely rejected or remapped Gilligan does not need to delete any code as it would not have been copied to the target system in the extraction phase. For instance, in Figure 2.5 `AEMonitor.enter()` is in a completely rejected class that was not copied to the target project. In contrast, `SpeedGraphic.getInstance()` is a rejected method within an accepted class; our tool removes this method during this step. The import statements are also updated in this step to reflect changes in dependencies due to removals.

8.2.2.4 Finalizing source code modifications

Each of the changes made by the three previous steps were made to the AST representation of the code, not directly to its text. This separation reduces the chance that one change will cause another alteration to fail. By working with an abstraction of the source code, the edits Gilligan makes do not cause shifts in the source code tokens that the AST cannot compensate for. For example, if we had injected a method fragment into the beginning class but then tried to remove a different rejected field subsequently, we could run into problems if we were to trace the model representation to textual locations (such as line numbers and character offsets); by manipulating the AST exclusively, we avoid these potential pitfalls. After all the steps are complete, Gilligan applies the changes to the files and writes them to disk, collecting statistics about the scope of the changes it has made. This final step can also apply any changes appropriate for the end of the process such as formatting the source code.

8.2.3 Supporting iterative planning and enactment

Figure 8.3(a) depicts the complete Gilligan workspace for a developer who is working on the scenario from Chapter 4. This figure shows two major differences compared to the last prototype (see Figure 7.7):

1. The problems view has been moved to be a part of the Gilligan perspective. This allows the developer to keep constantly appraised of the state of their reuse task in terms of compilation errors. In Figure 8.3(a) the developer can see that they currently have 36 compilation errors.

2. The validation view on the right side of the screen in the middle shows to the developer which dependencies, if any, they should investigate next.

The validation view is split into two parts. The left side of the view provides a list of the dependencies the developer still needs to investigate. When the developer selects any of these elements the right side of the view shows which structural elements, from the set of elements they have already accepted, are transitively dependent on the selected structural element. This gives the developer some idea of how important an element may be to their reuse plan; this is the only way that Gilligan violates DG 2.1, as in essence the right side of the validation view is showing the incoming dependencies for the selected element. If the developer clicks on any element in the right panel they are taken to that element in the structural views above. Gilligan populates the validation view by traversing the transitive closure of every accepted structural element and adding any element that has not had a decision applied to it to the validation view. While simple, this approach effectively alerts the developer to any node that is in some way relevant to something they want to reuse but about which they have not made a decision [**DG 3.1**]. These can be thought of as inconsistencies in the plan. Gilligan can most effectively enact the plan if it has been completely filled out; by using the validation view developers can ensure that their plan is as complete as possible. This view also can give them an indication that they have thoroughly investigated their reuse plan [**DG 3.1**].

A number of interesting observations can be made about Figure 8.3(a):

- The untriaged elements in the indirect dependency structural view, validation view, problems view, and undecorated source code in the source code editor are all related.

- None of `Colors.blues`, `Colors.BLUES_DARKEST` and `Colors.BLUES_MIDLIGHT` have been adorned with a decision colour in the source code editor. This is because the developer has not tagged any `Colors` fields with decisions yet; this is reiterated by the bottom three errors in the problems view along with the bottom two visible elements in the left half of the validation view.

- `COConfigurationManager` comprises the top three errors in the problems view. Not coincidentally, the top three structural elements in the left half of the validation view correspond to `COConfigurationManager` methods. These three errors arose because somewhere in the

reused code `addParamater(..)`, `getIntParamater(..)`, and `removeParamater(..)` are called but they are not marked as accepted, rejected, or remapped. These three dangling pointers would be fixed if the developer tagged these elements.

- In the indirect dependency view, the `ValueFormatter` interface has no decision applied to it. By selecting it, the developer can see that two classes, five methods, and one field are dependent on this interface. This gives them a strong indication that the interface is important to the source code they are investigating and that they should consider accepting this dependency.

After the developer has rejected `COConfigurationManager` from their reuse plan, their validation view updates, as does the error list (Figure 8.3(b)). The developer can now see that their error count has decreased, the error messages caused by `COConfigurationManager` have been resolved, and that `ValueFormatter` and `ParameterListener` should be considered next. The developer can keep working through these errors one at a time to see their effect. For instance, if the developer instead chose to accept `COConfigurationManager`, their error count would more than double, to 79 (Figure 8.4). In this way the developer can investigate alternative reuse decisions, and see how these decisions are reflected in the source code, in order to make the most informed decision possible.

8.3 Evaluation

Our previous evaluations investigated whether industrial developers perform pragmatic reuse tasks (Chapter 4), how effectively industrial developers could create pragmatic reuse plans (Section 6.3), and how much more effectively developers could locate structural dependencies using our tools (Section 7.3). Here, we wanted to answer three main research questions:

1. How much effort can semi-automating the enactment of a pragmatic reuse plan really save a developer?
2. How many fewer errors must a developer manually fix when performing a pragmatic reuse task with our semi-automation?
3. How does semi-automating the enactment of pragmatic reuse plans affect a developer's productivity when they are performing pragmatic reuse tasks?

To address these questions, we undertook two investigations. In the first we performed a case study by performing two pragmatic reuse tasks using both manual techniques and Gilligan. We then compared the minimum possible effort required to complete the pragmatic reuse task for the two treatments, where no delays or errors due to comprehension were considered; this represents the optimal scenario for each treatment.

Our second evaluation involved a controlled experiment using eight developers experienced in developing and modifying medium- to large-scale Java applications. Each developer attempted to

complete two pragmatic reuse tasks, one via the manual treatment and one via the Gilligan-based treatment. Participants were provided with a pragmatic reuse plan that had been created by us.

In both evaluations, "completion" was defined as the successful execution of a test suite that we provided for the purpose. One test suite was implemented as an Eclipse plug-in, while the other was a standalone application; we henceforth refer to both as test harnesses.

8.3.1 Task descriptions

Both evaluations used the same two tasks. Each of these tasks involved extracting specific functionality from an existing system and integrating it into a new system. Once integrated, the reused code needed to function within a test suite for the reuse task to be considered a success. Each task operated on a different open-source Java system from a different domain.

8.3.1.1 Metrics lines-of-code calculator

The Metrics Eclipse plug-in[2] can compute 23 different metrics (e.g., lines of code, cyclomatic complexity, efferent coupling, etc.) for resources inside Eclipse projects. This plug-in contains 229 classes comprising 14.5 thousand lines-of-code (kLOC). The goal of this task was to reuse the lines-of-code (LOC) calculator from this project; however, the system was not designed to enable individual metrics to be reused without the remainder of the Metrics plug-in.

The reuse plan for this task involved reusing portions of 8 classes. Successful completion of this task involved reusing 392 LOC. For the task to be a success, the reused code had to compute the LOC for every class in every project in the Eclipse workspace when activated by the test harness.

8.3.1.2 Azureus network throughput view

Azureus[3] is a client application for the BitTorrent peer-to-peer file-sharing protocol. Azureus contains 2,257 classes comprising 221 kLOC. It contains a view that visualizes its network throughput for the user. The goal of this task is to extract this network throughput view from Azureus and integrate it into a new system. This feature was not designed to be reused outside of Azureus.

The reuse plan for this task involved reusing portions of 6 classes. Successful completion of this task involved reusing 366 LOC. To succeed at the task, the reused code had to be able to correctly display a data set provided in the test harness.

8.3.2 Analysis of minimum required effort

Before performing an experiment to determine how automating the enactment of pragmatic reuse plans affects real developers, we wanted to understand the potential savings that this automation affords.

[2]http://metrics.sf.net v1.3.6
[3]http://azureus.sf.net v2.4.0.1

We wanted to answer two main research questions:

1. How much more work is it to perform a pragmatic reuse task manually compared to the same task when Gilligan semi-automates its enactment?
2. How many fewer compilation errors must a developer consider when their pragmatic reuse plan has been semi-automated compared to doing it manually?

To do this, we performed an case study that analyzed the effort an idealized developer would have to expend to perform two pragmatic reuse tasks using both Gilligan and performing them manually. In this case study we performed each task many times, with both treatments, to find the best way to perform the task. We then considered how much effort a developer would have to expend with both treatments if they performed the task the best way that they possibly could.

For each task we started with the reuse plan and two projects: the original project and the target project (containing the test harness). For the manual treatment our first action was to copy each of the classes involved in the reuse task from the source system to the target system. For the automated treatment our first action was to press the "enact plan" button in Gilligan. The number of compilation errors present after these operations is provided in Table 8.1. These error counts represent the size of the task facing the developer as they begin the task; we consider this an important indication of how daunting the task may seem at first.

Case	Gilligan	Manual	Error reduction
Metrics	3	62	95.2%
Azureus	11	32	65.6%

Table 8.1: Compilation errors for each task and treatment.

Compilation errors alone are not always a good indicator of required effort. We record the amount of work to enact the reuse plan in terms of the number of "edits" a developer must undertake to resolve all the compilation errors in the reused code. While this cannot attest to the functionality of the code, this gives us an indication of the relative amount of work required to get the code from both approaches to a consistent state for comparison. An edit involves making one conceptual change to the source code; if one edit resolves many problems, it still only counts as one change (for instance, if adding one import statement resolves five compilation errors, only one edit is counted). An edit represents a single conceptual change the developer makes to the source code. For example, in the Metrics system, the reuse plan specifies that calls to `AEMonitor.exit()` are to be removed from the target system. Although there are several of these calls to remove, doing so only counts as one edit as it is one conceptual operation. The minimum number of edits required to successfully complete each task with each treatment is given in Table 8.2. We performed each task multiple times (with the same plan for each treatment) to ensure that our edit counts were as low as possible.

Case	Gilligan	Manual	Edit reduction
Metrics	2	60	96.7%
Azureus	4	25	84.0%

Table 8.2: "Edits" required for each task and treatment.

Some edits require more thought and investigation on the part of the developer to resolve than others. These difficult edits arise due to conceptual mismatches between the original and target systems [Garlan et al., 1995]. For the Metrics task, only one of the edits represented a conceptual mismatch that arose from removing the reused code from the system for which it was designed. This occurred because the reuse plan specified that the `MetricsPlugin` class, and all of the elements it declares, should not be reused; this caused a dangling pointer in the `TypeMetrics.getCalculators()` method as it invoked the `getDefault()` and `getCalculators(String)` methods on the `MetricsPlugin` class. The solution to this problem is shown in Figure 8.5. Instead of simply commenting out some code, the developer had to think about what `getCalculators(String)` was actually doing and mimic this functionality within his own system, within the constraints of his reuse task (he did not want to return a set of calculators but only the one he was reusing, `TotalLinesOfCode()`); this short snippet relieved him from having to reuse the `MetricsPlugin` class. A complete description of each edit required to perform this task manually is included in Section E.1.1.

Three edits in the Azureus task represented conceptual mismatches; these were common between the two treatments. While Gilligan does not resolve any of the conceptual mismatch errors, it helps the developer to quickly identify them by resolving all of the trivial compilation errors that occlude them. Azureus uses an array of colours (called `blues`) to represent the various shades of blue used in the graph. When the graph was reused, its references to the Azureus `Color` class were remapped to the colour constants used in the target system. As the target system used different constants for different blues (e.g., `light_blue`, `dark_blue`, etc.) the developers would need to change the references on the arrays of `blues` to the various constants in the target environment. Other than these 3 edits, the remainder of the changes were mechanical compilation error resolution. A complete description of each edit required to perform this task manually is included in Section E.1.2.

8.3.3 Task effectiveness experiment

In this experiment we had one main research question:

- What is the difference, in terms of productivity, of a developer enacting a pragmatic reuse plan manually compared to using Gilligan's semi-automated enactment functionality?

To gather evidence to answer this question we performed an eight-developer controlled experiment. Four participants were industrial developers (I1 through I4) and four were experienced soft-

ware engineering graduate students (G1 through G4). All participants were experienced in developing and modifying medium- to large-scale systems written in Java, and in the use of the Eclipse IDE; relevant experience varied between 6 years for the least experienced participant (an industrial developer) and 12 years for the most experienced participant (also an industrial developer).

For this experiment we used the same two tasks as in the previous case study; these tasks are detailed Section 8.3.1. Participants were randomly assigned task–treatment pairs although we balanced the assignments by ensuring that each task–treatment pairing was completed by two graduate students and two industrial developers and to ensure that each task–treatment pair was performed an equivalent number of times. We ensured our experiment was balanced in this way because of the small number of trials involved.

Each participant used Gilligan for one task and performed the other task manually. The order in which the participants performed the two treatments was also randomized. We created a reuse plan for each task and provided identical versions for each treatment; the contents of these reuse plans are included in Figure E.3 and Figure E.4. A time limit of one hour was set for each task; we chose this time limit as it seemed like a reasonable amount of time for a developer to invest in this kind of task. We recorded whether or not the participants succeeded or failed for each task, how long they spent performing the task, and collected their final code for later analysis. After completing both tasks the participants completed a follow-up questionnaire (the questionnaire is included in Section E.2, p. 172).

The results of the experiment are shown in Figure 8.6. The figure depicts successful task–treatment pairings in green (diagonal hatching in greyscale). Those task–treatment pairings that were failures are indicated in red (solid in greyscale). The graph clearly shows that the participants successfully completed more tasks using Gilligan (8 out of 8) than with the manual treatment (4 out of 8). It is also clear that, for these tasks, developers were able to complete the tasks in less time using Gilligan than when undertaking them manually.

8.3.3.1 Manual treatment

The four participants who manually enacted the Metrics LOC reuse task were the least successful. By examining their resulting code and reading their comments in the questionnaire, it became clear that they knew there was a tricky problem they needed to fix somewhere in the reused source code, but they could not identify where in the source code it was. At the outset of this manual task, each participant (I2, I4, G1, G3) had 62 compilation errors to resolve; in the process of resolving these errors, they seemed to become disoriented. While each of them ended up with code that compiled, only I2 successfully completed the task (in 47 minutes). One of the participants became so frustrated with this task that after 24 minutes he gave up. One of the participants who failed at this task (I4) reported, "The manual approach was mostly drone work; it took longer to get the target project into a state where interesting problems could be solved." Even I2, who was successful, stated, "[The manual

task was] not hard, very tedious though. I was sitting there going 'this should be automated.'"

The other 4 participants undertook the manual version of the Azureus task. Only one of these developers (I1) failed to complete the task (after 60 minutes); the rest managed to finish in an average of 40 minutes.

8.3.3.2 Automated treatment

Each of the participants who undertook the Metrics LOC task using Gilligan managed to successfully complete the task (in an average of 19 minutes). In the questionnaires, these developers mentioned that they were able to concentrate on the 3 compilation errors that remained after Gilligan ran. Since two of these errors were trivial, they were able to focus on the single remaining error (which was a conceptual mismatch). While this error was tricky to solve, each of them was able to get the code to work successfully with the test harness.

All the participants also completed the Azureus task using Gilligan (in an average of 10 minutes). These developers did not seem to have any trouble changing the reused code to use the specific fields for blue rather than the `blues` array. For this task, I4 said, "There were still some syntactic mismatches, but what I found was that the problems that remained were more directly related to the actual misalignments between use contexts; they were more directly related to the reuse I was trying to achieve." I4 also stated that, "The simplicity of push-button enactment of the plan was a big win for Gilligan. I found it took me closer to where I wanted to be, and much faster. The manual approach was mostly drone work and took a lot longer to get [the] target project into a state where interesting problems could be solved. The manual approach also opened holes allowing errors in changes and omissions as well." I2 also made two short statements about the automated enactment, "automation makes the whole process worthwhile" and "the automatic task was great, hit the button, go, and it's immediately obvious what the tool did."

8.4 Discussion

In the first evaluation, we found that even if a developer were to manually undertake a pragmatic reuse task perfectly, expending a minimum amount of effort, he would still have to perform significantly more work than a developer undertaking the same task with Gilligan. For our two tasks the developer would need to consider and perform at least 3 times more edit operations than the same developer using automation support.

In the second evaluation we found that with automation support, developers were more likely to successfully complete our two pragmatic reuse tasks. While 4 of the 8 manual treatments ended in failure, all 8 of the automated treatments were successful. These successful tasks were also completed in much less time than their successful manual counterparts (40% of the manual time in the Metrics task; 25% of the manual time in the Azureus task). While we expected that Gilligan's semi-automation would save developers time, we were *very* surprised about the differences in success rates between the

two approaches. In the case study (Section 8.3.2) we identified several specific compilation errors that arose from integrating the source code into a new context that were harder to fix than usual. It was these errors that impeded the developers from succeeding at the manual tasks; when undertaking the tasks manually they would have so many errors to fix that they would either fix them simplistically (and thus fix the hard problems incorrectly without noticing) or they would become overwhelmed and skip over the hard problems when they encountered them because they "seemed hard" and carry on trying to fix all the easy problems first. By semi-automating the pragmatic reuse plan Gilligan fixed the easy problems automatically, enabling the developer to focus on only those tricky problems that remained.

Several participants commented that manual enactment was largely "drone work" (I4), "tedious" (I2), and "error-prone" (G4). The primary reason that participants failed to complete their tasks was because they did not resolve the conceptual mismatch between the reused feature and its new environment. In fact, in 3 of the 4 failure cases, the participants could not even identify where this mismatch took place, let alone resolve the problems created by the mismatch (G1 found the mismatch but could not resolve it). For the Gilligan-supported task–treatment pairings, all the participants were able to both identify and resolve the mismatch while successfully completing their tasks.

Two of the industrial developers made specific comments about how automating the enactment of the reuse plan positively changes their likelihood of using such a tool. I2 stated, "I wouldn't use Gilligan on its own; I have a bias against visualization and modelling tools. I don't care that the model is nice, it's running code that counts. The automation gives me the bridge I need to make the model valuable." while I3 said that, "I'm not one who likes planning for planning's sake. However, if the planning tool happened to also do the WORK for me, well then I would be much more likely to use it." We were encouraged to hear industrial developers becoming much more open to expending effort creating pragmatic reuse plans given our tool support for performing the bulk of the enactment task for them.

These two evaluations have focused entirely on evaluating DG 3.2. The case study and experiment both provide evidence that automating the enactment of the pragmatic reuse plan can save the developer time, effort, and make them more productive while they are performing pragmatic reuse tasks. How validation [**DG 3.1**] and improvements to support iterative development of pragmatic reuse plans [**DG 3.3**] affect the outcome of these tasks has not been evaluated; to do so requires developers to both plan and perform pragmatic reuse tasks (see Section 9.1).

8.4.1 Limitations

Our first evaluation considered the "ideal" developer who could complete his tasks with the minimum amount of work. While this developer is also not representative, he does represent the lower-bound that the best developers could strive to achieve. Even this developer had to perform considerably more work with the manual treatment than with the semi-automated one.

For the experiment, the number of participants was fairly small, at only eight. We have not attempted to quantify the relative effort of the treatments with statistical significance, therefore; we have reported times to give a sense of scale and trends. The trend in the results is unambiguously in favour of Gilligan.

The participants cannot definitively be categorized as representative of the intended target population (i.e., industrial developers familiar with Java development, the Eclipse IDE, and pragmatic reuse tasks). We used graduate students to increase the number of participants, due to the difficulty in recruiting industrial developers to evaluate a research prototype. We can see some differences between the graduate students and the industrial developers, each favouring the skills of one group or the other in differing task–treatment pairings. Most importantly, the trend in favour of Gilligan was uniform regardless of from which group came a participant.

8.4.1.1 Representativeness of tasks and systems

Only two tasks were performed on two systems. The tasks and systems cannot be definitively categorized as representative of all systems and all pragmatic reuse tasks. However, the tasks were non-trivial, being taken from real development scenarios and not synthesized for the sake of the research; each involved the reuse of functionality that had not been designed to be reused in the way we needed. Likewise, the systems are of medium- to large-scale from two disparate domains. There is nothing to suggest that these tasks and systems are otherwise special. We do suspect that "data-driven" applications are less likely to be amenable to the structure-oriented planning and enactment on which we have chosen to focus to-date. Whether our entire pragmatic reuse approach can be adapted to data-driven applications, remains to be investigated.

For the sake of experimental control, we provided the participants with pragmatic reuse plans for their tasks. Whether this reflects how Gilligan should be used in practice (i.e., one person planning the task and another person enacting it) is unclear. We would consider not having planned the task to be an impediment to enacting it; nevertheless, the use of Gilligan uniformly allowed this impediment to be overcome. It remains to be investigated whether having the same person planning a task and enacting it manually would result in better performance than the use of Gilligan. We conjecture that Gilligan would still outperform manual enactment, due to its ability to "abstract away" trivial compilation issues.

8.4.1.2 Net cost of pragmatic reuse

One might question how much effort must be expended to create or to interpret pragmatic reuse plans, and whether this effort would overwhelm the reported benefits of semi-automated enactment. In our experiment, each pragmatic reuse plan required less than 30 minutes to construct by us, despite our lack of experience with the original systems. As for interpreting the plans, Participant I3 states, "Ultimately [the Gilligan-supported treatment] seemed a bit like cheating, because I didn't really need

to know much about the reuse plan at all, it just went ahead and did it, and I fixed a couple of spots. I'll leave out the debate as to whether or not this is a good thing, but [it] did take me a LOT less time than the manual step." Most importantly, the tendency to trust the plan and to focus on the details of the remaining errors was the strategy that led to success. The incompleteness of the plans always leads to some errors, so the developer must investigate and understand the (isolated) problems. Our plans were also error-free, to the best of our knowledge, though necessarily incomplete. Given our relative ease in creating the plans, we feel that the lack of errors in the plans was realistic for these tasks; however, we must consider both the abilities of developers to plan as well as perform the pragmatic reuse tasks to see if the costs associated with creating the plan outweigh the benefits of the semi-automation of its enactment (this is discussed in Chapter 9).

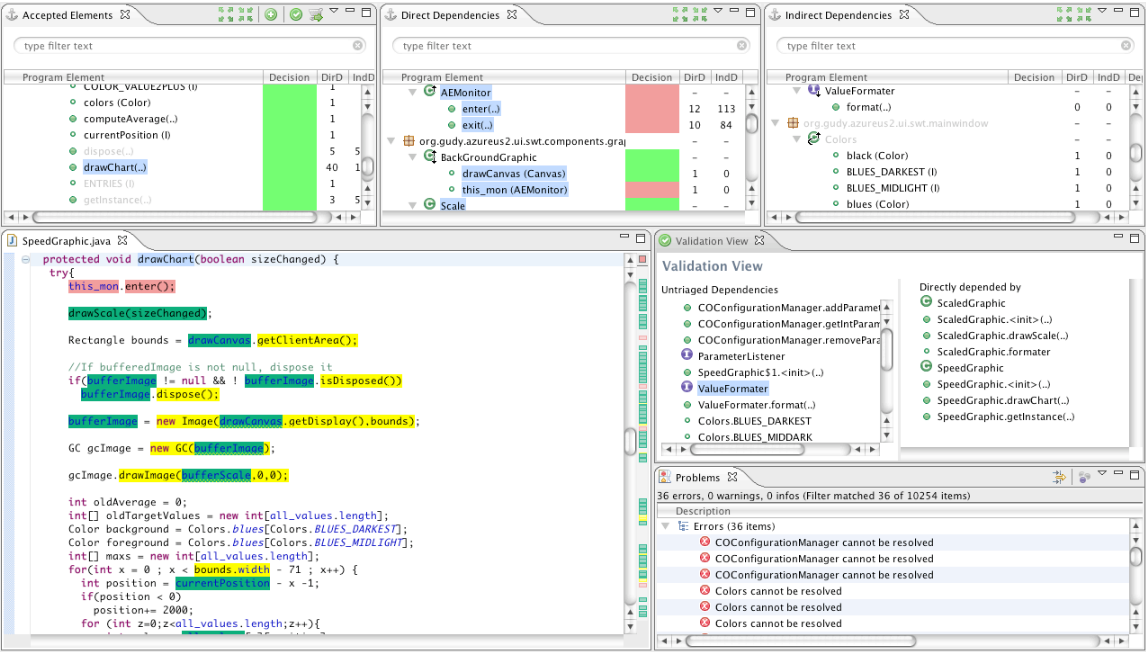
8.5 Summary

Compared to our first two Gilligan prototypes that focused on pragmatic reuse planning, our third prototype instead investigated techniques that could semi-automatically extract the source code specified by the developer in the reuse plan and integrate it into their project. This prototype added three new design goals. The first enabled the developer to validate their reuse plan so they could both tell when they were done their investigation, as well as to identify additional structural elements they should investigate. The second involved semi-automating the enactment. The third design goal revolved around features that encouraged iterative planning and enactment.

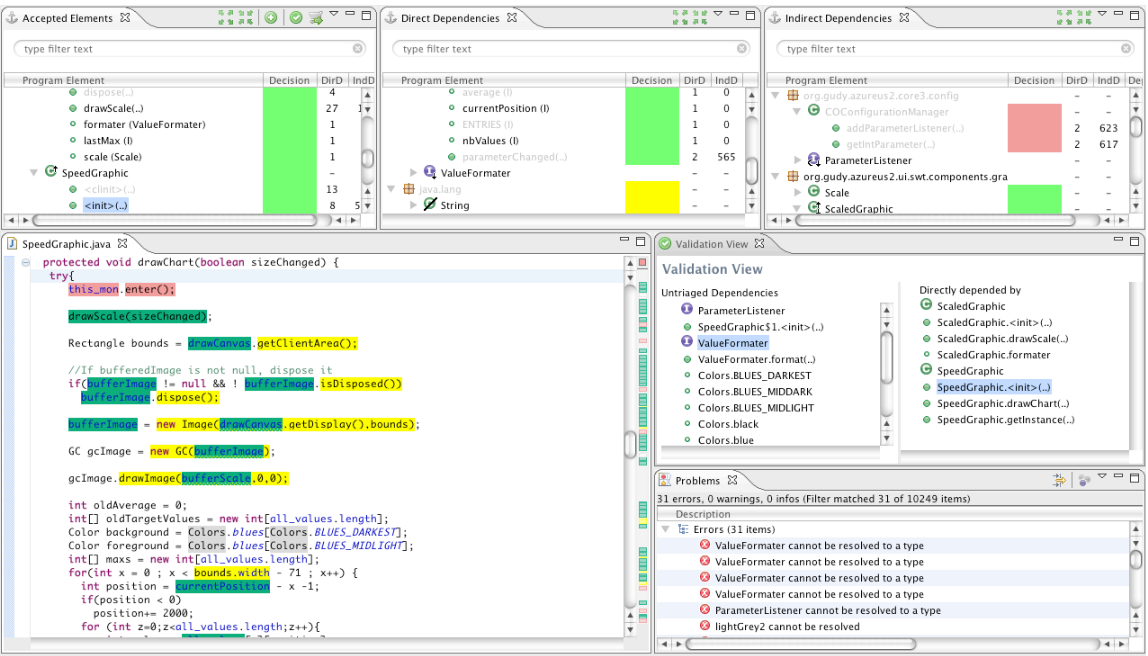
We evaluated this prototype using a case study and a controlled experiment. From these two evaluations we made four main observations:

- By automating the enactment of pragmatic reuse plans, Gilligan can considerably reduce the effort required to enact these plans.
- Gilligan can automatically resolve the majority of compilation errors that arise in source code that has been removed from its originating context according to a pragmatic reuse plan.
- Semi-automating the enactment of a pragmatic reuse plan has the potential to save the developer a significant amount of time.
- Semi-automating the enactment increases the likelihood that the developer will successfully complete a pragmatic reuse task by allowing them to focus on the problems that are important impediments to their successful completion of the reuse task.

Based on our observations of the participants working with this final Gilligan prototype, we decided to perform a larger end-to-end experiment in which the participants had to both plan, and perform pragmatic reuse tasks.



(a) Full Gilligan environment configured for iterative planning and enactment showing errors caused by dangling references, the validation view, and missing source code highlighting.



(b) Updated validation view after rejecting `COConfigurationManager`.

Figure 8.3: Third Gilligan prototype configured for iterative planning and enactment showing errors caused by dangling references, the validation view, and missing source code highlighting.

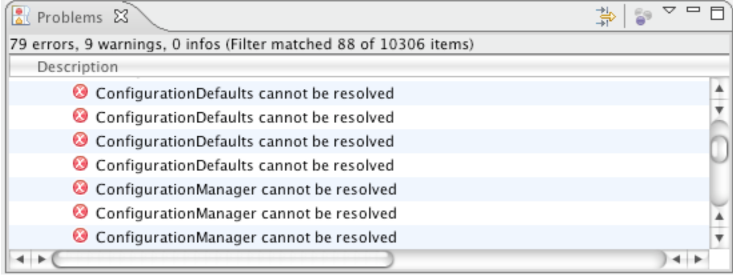


Figure 8.4: Updated validation view after accepting `COConfigurationManager`; this decision caused the error count to double.

Original code with dependency on `MetricsPlugin`*:*

```
return MetricsPlugin.getDefault().getCalculators("type");
```

New code that only returns the calculator that the developer wants to reuse:

```
Vector calculators  = new Vector();
calculators.add(new TotalLinesOfCode());
return calculators;
```

Figure 8.5: Snippet to resolve mismatch in Metrics task.

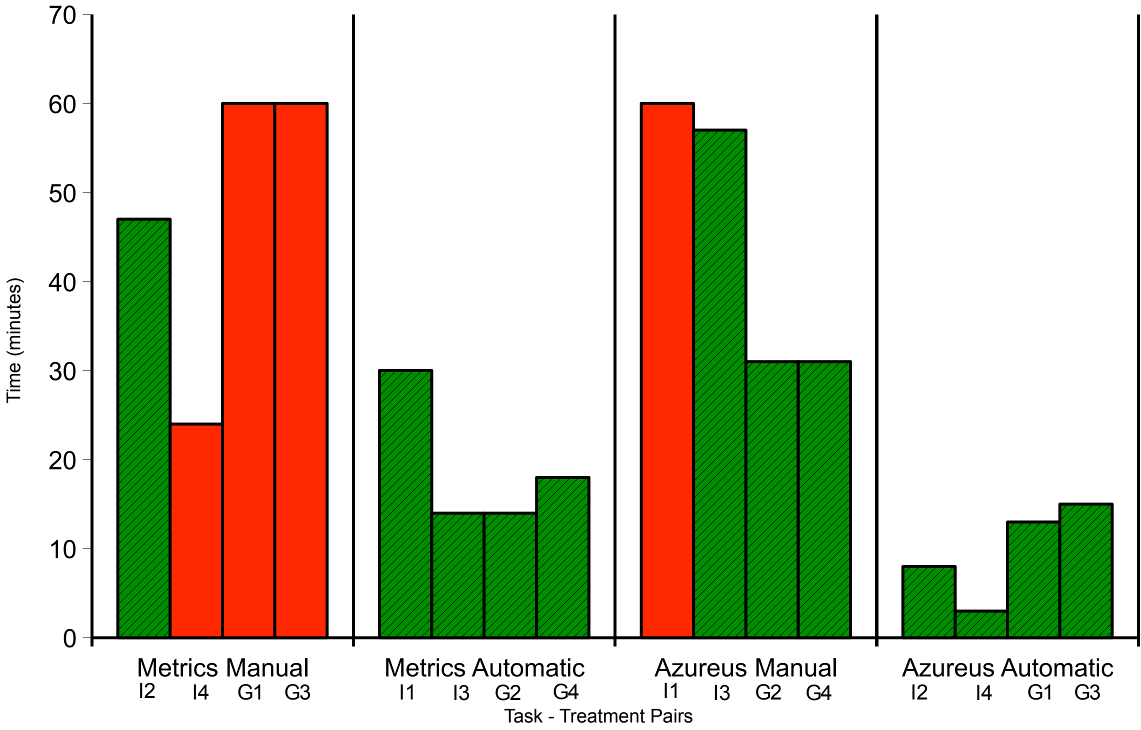


Figure 8.6: Results of the task effectiveness experiment. Green/hatched bars indicate success. Red/solid bars indicate failure.

Chapter 9

Holistic evaluation

Thus far in this dissertation, we have discussed five evaluations examining pragmatic reuse tasks, and how Gilligan can help with these tasks. These evaluations have examined eight main research questions:

① Do industrial developers actually perform pragmatic reuse tasks? (INDUSTRIAL SURVEY, CHAPTER 4) In this survey we found that industrial developers frequently perform pragmatic reuse tasks and have access to a large body of source code they can reuse in this fashion.

② What issues do industrial developers consider while evaluating a pragmatic reuse task? (INDUSTRIAL SURVEY, CHAPTER 4) Understanding the extent of the coupling between the source code they want to reuse and its originating system is a key impediment to determining the risk associated with a pragmatic reuse task.

③ Can industrial developers create pragmatic reuse plans using our graph-based Gilligan prototype? (INDUSTRIAL CASE STUDIES, SECTION 6.3) In this case study, our industrial participants found that they could successfully create pragmatic reuse plans for tasks derived from their work environment.

④ Do industrial developers perceive any derived benefit from planning their reuse tasks using Gilligan? (INDUSTRIAL CASE STUDIES, SECTION 6.3) This case study showed that our industrial participants felt that they had a better understanding of their task after creating a pragmatic reuse plan, compared to what they would have understood investigating the task manually.

⑤ Can developers more effectively locate structural dependencies using the tree-based Gilligan prototype compared to their normal practice? (CONTROLLED EXPERIMENT, SECTION 7.3) In this experiment we found that participants using Gilligan were able to identify 87% of relevant structural dependencies for a set of given tasks while participants performing the same tasks manually could only identify 45% of the relevant dependencies.

⑥ How much effort can Gilligan save a developer by automating the enactment of their pragmatic reuse plan? (CASE STUDY, SECTION 8.3) In this case study we found that, for two tasks, Gilligan reduced the number of conceptual decisions they would have to make by at least 84%.

⑦ How many fewer errors does a developer have to manually fix when performing a pragmatic reuse task whose enactment has been automated by Gilligan? (CASE STUDY, SECTION 8.3) In this case study we found that, for two tasks, Gilligan reduced the amount of compilation errors a developer would need to resolve by at least 65%.

⑧ How does Gilligan's automation of the enactment of pragmatic reuse plans affect a developer's productivity when they are performing these tasks? (CONTROLLED EXPERIMENT, SECTION 8.3) In this experiment we found that participants enacting pragmatic reuse tasks using Gilligan took 60% to 75% less time and were much more likely to succeed compared to participants performing the same tasks manually (8 of 8 successful trials compared to 4 of 8).

While each of these evaluations has added to the accumulation of evidence showing that industrial developers do perform pragmatic reuse tasks, and that Gilligan can help developers with various aspects of these tasks, none of the evaluations have directly addressed the thesis of this dissertation in its entirety:

➠ By providing developers with a mechanism for creating and enacting pragmatic reuse plans, can we help developers perform pragmatic reuse tasks more quickly and with greater confidence?

This will be the subject of the final experiment described in this dissertation.

9.1 Hypotheses

Each of our prior evaluations contributed to the body of knowledge that drove and informed the refinement of our Gilligan prototypes. To substantiate the thesis of this dissertation, we performed one comprehensive, final evaluation in the form of a controlled laboratory experiment. This evaluation was more ambitious than all our previous evaluations because it was a complete, end-to-end evaluation: participants had to plan and perform pragmatic reuse tasks, starting only with a task description and seed element, and ending with a solution that passed an executable test harness.

We had three hypotheses that we wanted to validate or refute quantitatively:

H-1: Developers using Gilligan to plan and perform a pragmatic reuse task will complete their task in less time as compared to performing the task using standard IDE tools alone.

H-2: Developers using Gilligan to plan and perform a pragmatic reuse task will be more likely to successfully complete their task as compared to performing the task using standard IDE tools alone.

H-3: Developers using Gilligan will abandon an infeasible pragmatic reuse task in less time than developers using standard IDE tools alone.

Based on the difficulties we encountered when industrial developers performed pragmatic reuse tasks on their own systems (Section 6.3), we decided that for this investigation we would provide, and control for, the tasks that the developers would perform and the environment they performed them within [Pfleeger, 1994]. We validated our hypotheses using a two-phase controlled laboratory experiment. To gain further insight into how developers plan and perform pragmatic reuse tasks, we collected as much qualitative data as possible for later analysis. As we were evaluating for a large effect size, a moderate number of participants sufficed.

9.2 Participants

Sixteen developers participated in our study (twelve male and four female). These participants were selected from the Calgary area. Each participant was an experienced user of the Eclipse IDE and the Java programming language. Our participants had a varied background: seven were industrial developers with various levels of experience (from 1 year to 10 years), 2 were undergraduate students, and seven were experienced graduate students (with 2 years to 9 years of development experience). We tried to select a sample representation of participants to portray an industrial workplace with both novice and expert developers. Some of the graduate students had worked in industrial positions in the past. When asking about development experience, we stressed that the participants should report values corresponding to years of "skilled" development; as an example of skilled development, we asked them to think of how long they have understood, and correctly used, advanced language features such as exception handling mechanisms.

To be eligible to partake in our experiment, each participant had to consider themselves 'experienced' with both the Java language and with the Eclipse IDE. Ten of our participants indicated that they used Java with Eclipse on a daily basis while the remaining 8 had done so at some point in the recent past. Our participants were paid $20.00 for their participation in our experiment. Table 9.1 provides an overview of our participants and their experience.

9.3 Tasks

The participants performed realistic tasks chosen from deployed software systems. Each of the tasks consisted of a description that gave the participant insight into the functionality they were trying to reuse, why they were performing the task, as well as background information about the system they were investigating. The participants were also given a single structural element as a starting point for their investigation.

Participants were given a small hint for each task; the intent of the hint was to help the participants stay focused on their task and to simulate normal guidance that may be given by a supervisor

P	Role			Familiarity (1 - 5)		Years Experience	
	UG	G	I	Eclipse	Java	In Industry	Dev experience
P1		✓		5	5	3	7
P2		✓		5	5	2	9
P3		✓		5	5	–	2
P4		✓		5	5	1	10
P5			✓	5	5	1	6
P6			✓	5	5	2	5
P7	✓			4	4	–	2
P8		✓		5	5	–	2
P9		✓		4	4	1	6
P10			✓	4	4	7	13
P11			✓	5	4	3	5
P12			✓	5	4	1	5
P13	✓			5	5	–	5
P14			✓	3	5	10	7
P15			✓	4	4	1	7
P16		✓		5	5	–	5

Table 9.1: Overview of participant experience.

or co-worker in an industrial setting. The starting point was given to help control variations between participants from the outset of the experiment. The same information was given for each trial, regardless of the treatment the participant was employing for that trial.

The experimental tasks were not identical to one another; indeed, each system was from a different domain and none of the tasks are similar in the functionality they are reusing. While we may have been able to construct identical tasks by creating our own systems to perform the tasks on, we decided that it was important to validate Gilligan on real software systems. Additionally, by using independent tasks and systems we aimed to reduce the possibilities of learning effects between the experimental trials.

9.3.1 Phase 1

The two tasks involved in the first phase of the experiment were chosen to evaluate H-1 and H-2. These tasks were chosen as 'good' pragmatic reuse tasks that made sense and could be effectively completed in the experimental time available. The two tasks in the first phase were always performed before the one task in the second phase.

9.3.1.1 QIF parser (Q)

The Quicken Interchange Format (QIF) is a file format that Quicken, a financial management software package, can use to save and load financial data. In this task, the participants were given a

QIF file that they had to parse. The jGnash project[1] is an open source personal finance package that has the ability to parse QIF files. The participants were tasked with identifying and extracting the QIF parser from jGnash and integrating it into a project that contained the test file and an executable jUnit[2] test case that could validate that the reused code worked correctly. Participants were pointed to one specific structural element within jGnash to begin their investigation, the `QIFParser.parseFullFile(..)` method. The test harness, sample QIF file, and our own pragmatic reuse plan for this task have been included in Figure F.

jGnash consists of 34 kLOC, 404 classes, 26 interfaces, 1,405 fields, and 2,907 methods. Our successful reuse plan for this task consisted of accepting 10 classes and interfaces, 46 fields and 37 methods for a total of 960 LOC.[3] We selected reusing the jGnash QIF parser for inclusion in this experiment as this feature is relatively well modularized in the jGnash system; while it was not designed with reuse in mind, it is somewhat self contained. There is only one conceptual decision that needs to be made for this task to be completed successfully; if the `QifTransaction._account` field is rejected, many potential dependencies that are not relevant to this reuse task are avoided. If the participant tries to reuse this dependency then they become dependent upon many other irrelevant classes. We also selected this task because it is an example of a task that is very easy to accomplish with standard IDE tools; in this sense we expected that participants using the manual treatment would succeed at this task in less time than participants using Gilligan given their greater experience with standard IDE tools.

For this task, participants were only given one hint: "All you want to do is to parse a QIF file into a programmatically-accessible format; you do not need to reuse any other banking functionality." This hint was provided to try to keep the participants focused on the intent of their reuse task.

9.3.1.2 **Related artists** (RA)

The aTunes project[4] is a comprehensive, open-source music player and manager written in Java. One nice feature of aTunes is that when you play a song by a particular artist, the interface populates a small window with a list of related artists in which you might also be interested. In this task, the participant was tasked with reusing the functionality that retrieves the related artist list. They were given the aTunes source code and a project that contained an executable jUnit test case that could validate that the reused code worked correctly by asking for related artists and checking to see if a list of artists was retrieved. The participants were pointed to one specific structural element within aTunes to begin their investigation, the method `AudioScrobblerService.getSimilarArtists(..)`. The test harness has been included in Figure F.

aTunes consists of 33 kLOC, 302 classes, 17 interfaces, 1,237 fields, and 2,468 methods. Our

[1] http://jGnash.sf.net v1.11.6
[2] http://junit.org v4.4
[3] A complete description of our pragmatic reuse plan for this task is included in Figure F
[4] http://atunes.sf.net v1.6.0

successful reuse plan for this task consisted of accepting 10 classes and interfaces, 18 fields, and 31 methods for a total of 491 LOC.[5] We selected reusing the related artists feature from aTunes for inclusion in the experiment as this feature represented a much more difficult reuse case than the jGnash QIF feature. The feature itself is more complicated, is spread throughout the system much more, and is co-located with many other features within aTunes. While it is co-located with other features, it is not necessarily coupled to them and this means that the code appears to be more coupled than it actually is. In this task the participant must carefully reject many structural elements that are a part of the classes the participant wants to reuse but are not actually a part of the feature they want to reuse.

For this task, participants were only given two hints: "You are only interested in retrieving a list of related artists, other data such as related albums is not of interest" and "How well the reused code performs is not of concern for this task." The first hint was to remind the participants that a lot of the other co-located functionality was not relevant to their task; the second hint was because aTunes aggressively caches related artists and we did not want to require the participants to reuse the caching mechanism.

9.3.2 Phase 2

The third task was used only in the second phase of the experiment; it was chosen because we considered it to be a 'bad' task: not only could the task not be finished in the experimental timeframe, we do not believe any developer would really try to complete this task due to the degree to which the feature is coupled to its underlying system. This 'poison pill' was inserted to see how long it would take the participants to decide that the task was ill-advised and that they should give up. This phase was directed at testing H-3.

The third task was the main reason we split the experiment into the two phases: we were concerned that the participants may become overly pessimistic for the two 'good' pragmatic reuse tasks if they performed one of them after the 'bad' task, particularly given their lack of experience using Gilligan. At the same time, we hoped that the experience they gained in the first phase (with both treatments) would help them to better assess the suitability of the third task.

9.3.2.1 **Torrent downloader** (TD)

The Azureus project[6] is an open source BitTorrent peer-to-peer file transfer program written in Java. For the past several years, Azureus has provided a full-featured, robust UI for transferring files on the BitTorrent network. In this task, the participant was tasked with reusing the core BitTorrent file transfer functionality from Azureus without any of its UI elements. The participants were given the Azureus source code and a project that contained an executable jUnit test case that could validate that

[5]A complete description of our pragmatic reuse plan for this task is included in Figure F.
[6]http://azureus.sf.net v2.4.0.2

the reused code worked correctly. The participants were pointed to one specific structural element within Azureus to begin their investigation, the method `TorrentDownloaderImpl.run()`. The test harness and has been included in Figure F.5. Azureus consists of 222 kLOC, 1,657 classes, 582 interfaces, 5,219 fields, and 12,253 methods. We do not include our reuse plan for this task as we do not believe it is possible to create one.

For this task, participants were given two hints: "Do not worry about security-related aspects of the Azureus system", and "You do not need to reuse magnet functionality, just the basic torrent transfer feature." These hints were added to restrict the amount of code the participants had to consider reusing.

9.4 Experimental procedure

Each participant took approximately 3 hours to complete the experiment; during this time they completed 6 discrete steps:

1. *Orientation.* Upon settling in our experimental space we started (after allowing them to read and sign the requisite ethics forms) with a quick questionnaire to gain an understanding of the participant's development experience. The complete text of this questionnaire can be found in Section F.2 (p. 187). All the data in Table 9.1 was collected from this initial interview. We then gave the participants a quick verbal overview about the terminology we would be using during the study and explained what they would be doing during the study. The orientation took between 15 and 20 minutes.

2. *Training.* The training task had two phases: first we would quickly demonstrate Gilligan to the participant explaining the 3 structural views, how to make decisions about elements, how the source code highlighting worked, and how the validation and automatic enactment features worked. We then set the participants to work on a sample task. During this period they were free to ask as many questions as they wished. Once they felt that they understood how Gilligan worked we proceeded with the experimental trials. Participants spent between 15 and 20 minutes on the training task.

3. *Phase 1.*

   (a) *Trial 1.* The first trial was always one of the randomized trials for the Phase 1 tasks; the trials are described in Section 9.4.1. The time limit for this trial was 40 minutes.

   (b) *Trial 2.* The second trial was always one of the randomized trials for the Phase 1 tasks. The second trial always balanced the treatment with the first trial such that they never had the same treatment for both tasks.; the trials are described in Section 9.4.1. The time limit for this trial was 40 minutes.

4. *Phase 2.*

    (a) *Trial 3.* The third trial was always the trial for the Phase 2 task with only the treatment being randomized; the trials are described in Section 9.4.1. The time limit for this trial was 40 minutes.

5. *Debriefing.* Each participant completed an exit questionnaire reflecting on their experiences for each task and the different treatments. The complete text of this questionnaire can be found in Section F.5 (p. 193). The exit questionnaire took between 10 and 15 minutes to complete.

9.4.1 Performing a trial

For each trial, we first described the task to the participant and showed them the source and target projects. Participants could ask for clarifications on the task if they wanted any, although in the course of the experiment none of the participants asked for more detail than was already provided. We encouraged participants not to examine the test harness; we did not want the participants to investigate this in depth as there were additional clues in each test harness that we would rather they discover on their own while performing the task. Each trial involved 40 minutes of development time and up to 15 minutes of questioning. Participants were encouraged to talk to themselves aloud while they were performing their tasks so we could better understand their thought processes.

Participants were interrupted at specific intervals during their task to answer mid-task questionnaires; these questionnaires were the same regardless of the trial or the intervention interval.[7] The first questionnaire was given after 5 minutes; we chose 5 minutes because we wanted to see if the participants had any initial 'gut feelings' about their task. The second questionnaire was after 15 minutes, the third at the end of the task at 40 minutes. Participants did not have to answer any of our questions if they had finished a task before the interval for the questionnaire had happened. The time taken to answer the mid-task questionnaires was not deducted from the participant's total time (e.g., every participant was able to work on the task for the same amount of time, regardless of their level of verbosity).

Participants were able to ask us questions while they performed a task, but we only gave clarifications on the original task description. While performing a task, if the participant decided they could not, or would not, continue this task, they were allowed to proclaim that they 'gave up'.

After completing (or giving up) on a task, the participant was again asked a series of questions.[8]

9.4.2 Data collection

A large variety of data was collected while this experiment was underway; some of it was automatically recorded by our heavily-instrumented IDE environment while the rest was recorded by hand.

[7] The complete text of the mid-task questionnaires can be found in Section F.3 (p. 189).

[8] The complete text of the end-of-task questionnaire can be found in Section F.4 (p. 191)

All the questionnaires were treated as short 'interviews' to give the participants the most opportunity to provide detailed answers; if they said too much, we asked them to stop so we could catch up with our hand-written notes. We opted not to record the tasks either with video or audio for two reasons: (1) we wanted our participants to be as comfortable as possible; and (2) we did not think the added analysis burden was worth the effort over attentively-collected hand-written notes.

We took a variety of notes during the trials while the participants were working through their assigned tasks. We recorded: significant comments, questions they asked themselves aloud, comments they made, strategies of note, the number of editors open and compilation errors present at various intervals, major milestones, and specific problems they were having. The note-taking process was fairly flexible according to the specifics of the trial. Time stamps were recorded along with most of the observations. We also recorded time stamps according to when the task began, when we interrupted for each interview, when the participant resumed work on their task, and when they succeeded or gave up. The ultimate success or failure of the task was also recorded.

Our instrumented version of Gilligan recorded every structural element the participant navigated through, what decisions they made, when they requested to see an annotated source editor, and noted whenever they validated or enacted their plan. If the participants ever changed their mind about a decision this was also captured. The Eclipse IDE itself was further instrumented to include basic navigation data between various views and the source code editor. When the participants completed a task with Gilligan the reuse plan was automatically saved for later analysis.

After the participants had completed all three trials, we saved their evaluation workspace. This enabled us to further analyze their solutions for each task. Using their solutions we could see which structural elements they reused, which elements they did not, how many lines-of-code were involved in their solutions, and investigate any other specific question about the state of their solution.

9.5 Experimental design

In the first phase of the experiment we considered two factors:

- Tool: Gilligan (G) and IDE-only (M).
- Task: QIF Parser Q and Related Artists RA.

For both phases of our experiment, the tool factor had two levels: using Gilligan to plan and perform the pragmatic reuse task (called G) and the default manual case (called M), whereby the participants would use standard IDE tools.

The task factor for the first phase consisted of two levels, each of which were an independent reuse tasks, Q and RA.

Both the tool order and task order were counterbalanced to account for confounding effects introduced by the varying skill levels of our subjects and learning effects in the experiment as each

participant completed trials with both treatments their individual skills would be spread across all treatments.

A full within-participants factorial design involving these two factors would require participants to repeat both tasks twice, which would not yield valid results in our case, since they would already be familiar with the solution to the task they were repeating. Thus, each participant completed one task with Gilligan and the other with only the IDE. This design is best understood by considering the second factor to 'order', resulting in a mixed design on the following two factors:

- Tool: Gilligan (G) and IDE-only (M); within-participants.
- Order: Q and RA; between-participants.

The first phase of the experiment consisted of 32 trials: each of our 16 participants would perform 2 trials. The trials were assigned in four blocks B1A, B1B, B1C, B1D, each of which corresponded to a task–treatment–order tuple. These tuples are listed in Table 9.2. Participants were randomly assigned to a block although we balanced the assignments to ensure that each of the blocks was replicated an equivalent number of times (4 each). This first phase was designed to help us evaluate hypotheses H-1 and H-2.

Block Name	No. of Trials	First Task	Second Task
B1A	4	Q–G	RA–M
B1B	4	Q–M	RA–G
B1C	4	RA–G	Q–M
B1D	4	RA–M	Q–G

Table 9.2: Four blocks for the first phase of the experiment.

The second phase of the experiment used a between-subjects design on the tool factor. The two levels were Gilligan and IDE-only. In this phase, only one task was used (TD).

This phase of the experiment consisted of 16 trials: each participant only performed one trial in this phase. These trials were assigned to two blocks B2A and B2B, each of which just specified which treatment (G or M) was used. All the trials in the second phase always happened after the trials for the first phase had been completed. Each block was performed 8 times although we balanced the assignment to ensure that each B2 block came after each of the four B1 blocks 2 times. Again, this balancing was to account for learning effects from the first phase.

In accordance with Sadler and Kitchenham [1996], we tried to control for learning effects using techniques they recommend. Gilligan represents a sophisticated departure from our participants' regular development activities. Prior to beginning the experiment, each participant performed the same training task using Gilligan. While performing this task, they were free to ask any questions they wished about the tool and how it worked. Our alternative hypotheses for H-1, H-2, and H-3 have

all been arranged such that the alternative hypothesis is favoured by our participants' experience; they all have years of experience using standard IDE tools but will only have had minimal (less than 30 minutes') experience using Gilligan. We arranged the experiment this way so that if we did find an effect in favour of Gilligan it would not have been unduly influenced by a learning effect. To further minimize learning effects we randomized the orders of the tasks and treatments during the B1 blocks.

Each of the three tasks used by the two phases of the experiment used independent developed systems, shared no common code, and was from a different domain; this was to minimize the chances of any memory-effect interactions between the different tasks.

We used the mid-task and post-task questionnaires as a way to provide our participants with short breaks; this was to reduce fatigue effects that may have arisen from having them work for three hours on complex tasks.

9.6 Quantitative results

The quantitative results for each of our hypotheses are discussed in Section 9.6.1, Section 9.6.2 and Section 9.6.3. We were interested primarily in effects involving the tool factor, but not in the effect of task order (which was randomized to account for bias). We therefore analyze differences only for the tool factor.

9.6.1 H-1 analysis

To test hypothesis H-1 we first examined the amount of time the participants took while performing the block B1 trials. Figure 9.1 shows the relationship, for the jGnash and aTunes tasks, between the treatment condition and control in terms of time. For each column, the thick line inside each rectangle represents the median while the upper and lower bounds of the rectangle represent the 75th and 25th percentiles respectively; the rectangle represents the middle 50% of the data points. The upper and lower fences represent the maximum $1.5 \times$ inter-quartile range; points above the top fence and below the bottom fence can be considered outliers.

For the block B1 trials, there was only one outlier data point; this represented a participant who completed the jGnash task using Gilligan in only 5 minutes. We did not remove this outlier because the participant successfully completed the task; although they were fast they managed to create a functional solution. For five of the trials (one trial for jGnash–Manual and four trials for aTunes–Manual) the participant was unable to complete their task within the allotted 40 minutes; while none of these points were outliers, they represent numerical values that may have been higher if the participants had longer to finish their tasks. For both manual boxplots, this has caused the area of the boxplot to be more compressed than it might have otherwise been if the participant had been able to keep working on their task. As this compression was only present for manual tasks, difference could only have been greater if the participants had been given more time.

We ran a repeated measures factorial ANOVA on tool and task order. There is a significant

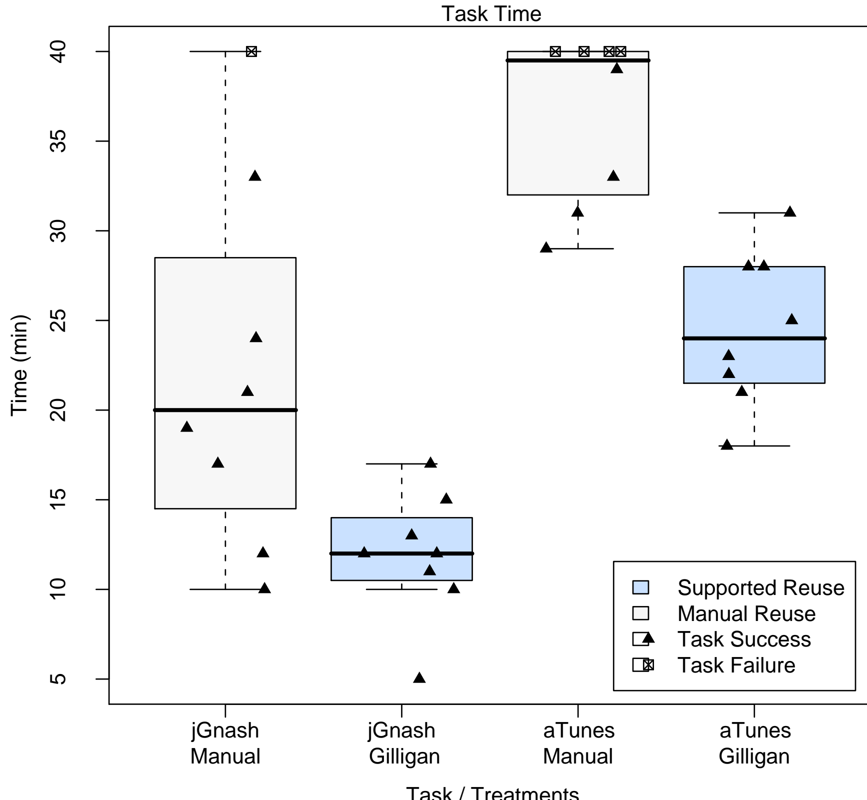


Figure 9.1: Block B1 time to completion by treatment. Data points on the blue bar columns represent trials that used Gilligan, the clear columns represent the control. Triangle icons represent successful tasks while boxes with an 'x' through them represent failures.

main effect of tool ($F(1,14) = 5.1, p = 0.04$). Participants were significantly faster using Gilligan ($M = 18.7$ minutes, $SE = 2.2$ minutes) than using no tool ($M = 28.8$ minutes, $SE = 2.7$ minutes). The main effect of task order was not significant ($F(1,14) = 0.9, p = 0.35$) nor was the interaction ($F(1,14) = 0.0, p = 0.94$). Because of this, we reject the null hypothesis with respect to H-1 and consider Gilligan to be a significant influence on the amount of time required to plan and perform a pragmatic reuse task.

9.6.2 H-2 analysis

To test hypothesis H-2 we performed Fisher's exact test; we chose this method as Pearson's Chi-squared test requires larger samples than we had available in this study. The contingency table used for this evaluation is given in Table 9.3. Participants succeeded significantly more often when using Gilligan ($p = 0.043$).

	Success	Failure	TOTAL
Manual	11	5	16
Gilligan	16	0	16
TOTAL	27	5	32

Table 9.3: Contingency table for success and failure compared to treatment.

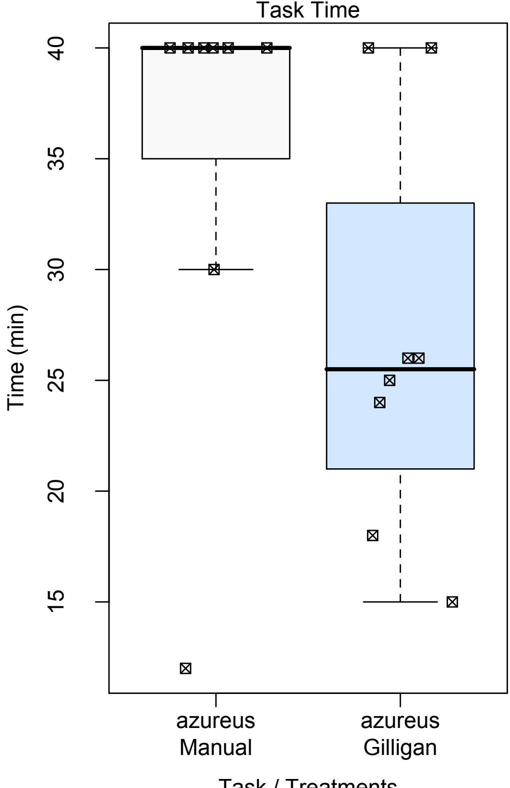


Figure 9.2: Block B2 time to completion by treatment. Data points on the blue bar columns represent trials that used Gilligan, the clear columns represent the control.

9.6.3 H-3 analysis

To accept or reject hypothesis H-3 we examined the amount of time it took for a participant performing a block B2 trial to abandon their task. This task had one outlier wherein a participant performing the manual case gave up after only 12 minutes. As this developer gave up not because they felt they had good reason to do so, but because they became overwhelmed during their investigation, we have chosen to eliminate this data point from our analysis. This decision is consistent with Chauvenet's criterion [Taylor, 1996] for identifying an outlier ($0.4 < 0.5$).

After eliminating the outlier data point, we applied the Mann-Whitney-Wilcoxon (MWW) test and received a statistically significant result ($U = 8, p = 0.012$).[9] We chose this conservative, non-parametric test, as phase 2 was a between-participants design, effectively reducing our sample size by a half. As we believe eliminating the outlier data point mentioned previously is appropriate, we use this p-value and reject the null hypothesis with respect to H-3.

It is also noteworthy that only 2 of the 8 participants performing the manual trials gave up on their experimental task before the 40 minute time limit while 6 of the 8 participants performing the Gilligan trials gave up during this time period. Anecdotes provided by the participants for this experiment are given in Section 9.8.2.

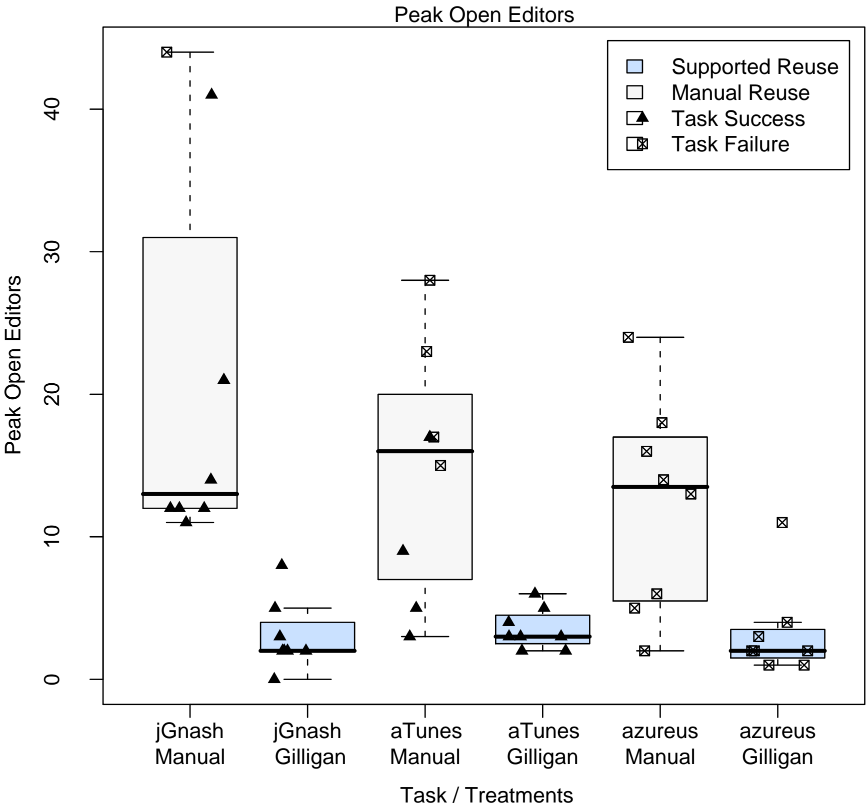


Figure 9.3: Peak open editors by task–treatment. Data points on the blue bar columns represent trials that used Gilligan, the clear columns represent the control. Triangle icons represent successful tasks while boxes with an 'x' through them represent failures.

[9] This test yields a less dramatic result if this outlier is considered ($U = 16, p = 0.072$).

9.6.4 Other quantitative analyses

While we found a huge variation along other metric axes (such as LOC and errors), one interesting metric we recorded was the peak number of source code editors a participant had open during their task. Figure 9.3 shows, for each task in both B1 blocks and B2 blocks the maximum number of editors the participant had open. We have not conducted statistical tests for this relationship, we show it only out of interest. This strongly confirms that developers performing Gilligan-supported tasks were able to work effectively with the abstract representation provided by Gilligan as they were successful, yet managed not to open a plethora of editors.

9.7 Qualitative results

During the final experiment 254 pages of hand-written notes were taken while the participants performed their experimental tasks; these notes primarily comprised of statements made by the participants but also included some observations. The notes were transcribed onto 90 type-written pages that were split into individual comments, thoughts, or actions. The resulting 954 comments and observations were analyzed using a grounded theory approach [Corbin and Strauss, 1990]. Grounded theory allowed us to group the comments according to their content using some coding criterion. We used an an open coding approach [Miles and Huberman, 1994] to assign codes to the collected data; by not pre-defining our coding strategy we allowed the categories to be iteratively developed and refined. The groups were identified by iterating on the comments four times. After the first iteration, 57 individual themes were delineated that contained between two and 60 comments. The second iteration shrunk the number of themes to 44 but further subdivided these into 136 sub-themes; during this process many sub-themes were merged and split as further commonalities and divergences were found. During the final phase, the 44 themes were grouped into 8 individual concept categories; each of these represented a high-level concept that unified its constituent themes. These concept categories comprised between 3 and 9 themes.

Five of these concept categories were *organic*— that is, they naturally arose from the data; the developers were not answering specific questions that were asked of them. These concept categories each pertained to a different aspect of exploring, analyzing, and performing a pragmatic reuse task. Three of the concept categories (including 16 themes and 66 sub-themes) were *prompted* as they were a result of both observation and participant answers to specific questions that were asked of them. Section 9.7.1 describes the organic concept categories while Section 9.7.3 describes the prompted concept categories.

An overview is provided for each concept category, along with its themes and sub-themes. For each theme the total number of comments given and the number of individual participants who gave them is enumerated as an indicator of the support for that theme. A synthetic quote is provided for each theme; the intent of this quote is to give a general understanding of what that theme represents. These synthetic quotes were generated by combining portions of the collected quotes and integrating

them into a single cohesive statement. A number of significant, interesting, or demonstrative quotes are also included to further reinforce each sub-theme.

It is important to note that each of these categories was derived from observations of developers performing trials using both the Gilligan treatment and using standard IDE tools.

Appendix G contains a detailed 53 page analysis of the qualitative findings from this experiment. In this section, we present an overview of these findings as well as several interesting specific points. For more data and additional analysis, please see Appendix G.

9.7.1 Organic concept categories

The five organic concept categories that arose from the card sort were both surprising and interesting. They do not map directly to the steps that developers undertake while performing pragmatic reuse tasks; rather they comprise a combination of actions and mental processes that developers must perform while considering these tasks. Each of the concept categories is intertwined; a short overview of each of them is given here, along with a quick discussion of how they are related. As the concept categories were derived from the data, not fit into pre-defined bins, they do not fit together in a way that supports a natural, linear progression through them. They are presented in an order that tries to be as natural as possible for the reader; however, forward referencing is unavoidable and each of these categories takes place concurrently and iteratively as the developer progresses through their reuse task.

The high-level concept categories are dependency identification (Section G.1.1), understanding (Section G.1.2), maintaining mental models (Section G.1.3), hypothesis testing (Section G.1.4), and performing pragmatic reuse tasks (Section G.1.5). These categories can be further collected into three groups: those pertaining to actions, those pertaining to understanding, and those that are a hybrid. The dependency identification and performing categories both involved the participant commenting on actions they were making to move throughout the source code. The understanding and mental models categories focused on the participant building an accurate representation of the source code so they could be effective in their task. The hypothesis testing category is a hybrid of these other categories: the participants were actively performing small-scale tasks to further their understanding of what they were doing.

Dependency identification. Dependency identification is a specialized form of information gathering. Participants in this phase were trying to identify what structural dependencies were in the system and where the code representing these dependencies existed. They would use the dependencies they found to build their understanding of the source code and translate this understanding into their mental model of how the system was structured and functioned. The methodologies the participants used to identify dependencies were remarkably different depending on the experimental treatment they were using for each task; in the manual treatment the participants spent most of

their time poring over the source code in a line-by-line fashion, while in the Gilligan treatment the developers quickly navigated throughout the system.

Understanding. The participants built their understanding of how the system worked and was structured by investigating the dependencies and the source code within the system. The treatments did not diverge very much from how the participants built this understanding; in both cases the participants would try to rely on the naming of the structural elements as much as possible to infer some element's functionality to avoid having to read the code in an in-depth manner. With both treatments the participants would sometimes need to resort to looking very carefully at the individual lines of source code, as the names alone were not enough. The need to understand was prompted by the developer identifying a dependency that seemed interesting; the participant would then incorporate this dependency into their mental model of the reuse task.

Mental models. Building an effective mental model was of critical importance for the participants to be successful in their reuse task; without a model of how the code worked that the participant wanted to reuse, they lost track of the details and ended up following false paths, getting lost, and making more work for themselves. Ultimately, Gilligan provided far greater support for encoding the decisions that the participants made while building up their understanding of the task and consequently the participants felt like they were much better able to focus on the task without being distracted while performing the Gilligan-supported treatments.

Hypothesis testing. Performing a pragmatic reuse task is a heavily iterative process. One of the main reasons for this iterative process was the testing of hypotheses. Once a participant had determined that a structural element was relevant to their reuse task they would attempt to reuse it by copying it to their system. They would then analyze the resulting errors and decide whether the errors induced by reusing the code outweighed the benefits of reusing the structural element or not. The results of this process both built their understanding of the system and contributed to their mental model. The treatment employed by the participants had a huge effect on the participant's willingness to investigate different hypotheses; manually this process was very difficult to do in an in-depth manner, while with Gilligan the participants could easily change their mind if necessary. While this could be viewed as a specialization of performing the pragmatic reuse task, its emphasis on developing the participant's understanding and building their mental model makes this category significant and independent.

Performing pragmatic reuse tasks. Themes related strictly to performing pragmatic reuse tasks fell into this concept category. Of the five concept categories this is the least surprising. These themes concern strategies and methodologies used by the participants in both treatments to turn their mental model into a complete pragmatic reuse task. The approaches used by the participants for the treat-

ments are unsurprisingly divergent, as Gilligan automated the majority of this work for the participant; however, Gilligan's support for rejecting structural elements had an impact on the participants and how they understood and built their mental models for their tasks.

Table 9.4 lists the 25 themes encapsulated by the 5 concept categories. Each theme is listed in the order they appear in the subsequent text. The table also reports the number of participants who made quotes that fell into each theme, the total number of quotes in the theme, and the theme's synthetic quote.

# Participants	# Quotes	Synthetic Quote
Dependency Identification		
16	176	
16	39	*It is frustratingly difficult to manually identify the relevant dependencies for a pragmatic reuse task.* ▶
8	10	*It can be overwhelming and disorienting to manually navigate through unfamiliar source code.* ▶
7	9	*Using Gilligan it is easy to locate and navigate through source code dependencies.* ▶
15	40	*It is important to be able to measure and track progress while performing complex reuse tasks.* ▶
14	29	*Gilligan provides the majority of the information necessary to make informed decisions about reuse plan alternatives.* ▶
14	32	*Gilligan's validation view can help developers avoid spending large amounts of time wading through lengthy lists of compilation errors.* ▶
7	15	*Being systematic helped developers feel that they were better able to manage large, complex tasks.* ▶
2	2	*Gilligan helped developers spend less time investigating irrelevant common dependencies.* ▶

# Participants	# Quotes	Synthetic Quote
Understanding		
16	58	
12	18	*The source code is the authoritative source of information. Gilligan reduces the burden of having to read the code but provides access to it as needed.* ▶
6	11	*Understanding the functionality provided by the source code is key to determining whether it should be considered for reuse.* ▶
4	5	*Manual investigations tend to be breadth-first while Gilligan enables a more explorative methodology.* ▶
14	24	*Poorly named or modularized source code can obscure its functional role from a developer.* ▶
Mental Models		
14	74	
9	16	*Gilligan explicitly encodes reuse decisions; this reduces overhead required to manually remember all of the salient details.* ▶
11	26	*By providing a high-level view of a reuse task, Gilligan makes it easier for a developer to get a global understanding of the source code they are reusing.* ▶
5	6	*Gilligan helps developers focus on their reuse task rather than being overwhelmed by numerous compilation errors.* ▶
6	7	*Gilligan helps developers manage the complex details that arise during pragmatic reuse tasks.* ▶
8	19	*Developers are encouraged and supported in making explicit, consistent, decisions using Gilligan.* ▶

# Participants	# Quotes	Synthetic Quote
Hypothesis Testing		
16	76	
12	32	*Developers frequently make poor decisions while performing pragmatic reuse tasks; these are difficult to reverse manually.* ▶
11	14	*Gilligan encourages and assists developers in investigating alternative reuse strategies.* ▶
12	25	*By helping developers progress through a pragmatic reuse task one decision at a time, Gilligan enables them to better track their successes and failures.* ▶
3	4	*Developers using Gilligan are more confident in the quality of their solutions.* ▶
Performing Pragmatic Reuse Tasks		
16	91	
14	20	*Most of the manual work required during a pragmatic reuse task is conceptually simple but labour intensive.* ▶
14	41	*Reuse tasks require less time, effort, and frustration when they are performed with Gilligan.* ▶
7	12	*Developers are likely to copy entire packages while manually performing a reuse task, even though they know it is likely to backfire on them.* ▶
8	18	*Being able to easily remove elements that are not related to the reuse task is essential to reusing only relevant source code.* ▶

Table 9.4: Overview of the organic concept categories.

9.7.2 Interesting themes and quotes

We have selected six of the themes outlined in Table 9.4 for further discussion.

It is frustratingly difficult to manually identify the relevant dependencies for a pragmatic reuse task. Before a developer can reuse a dependency, or even decide whether they want to reuse it or not, they have to know that it exists. In an earlier evaluation (Section 7.3) we found that developers had trouble locating dependencies manually; this was further reflected in the comments made by all 16 participants in this experiment. Two comments were emblematic of the problems the participants faced in the manual treatments: "Resolving the dependencies was hard: something that looked innocuous to reuse proved to be a nightmare." (P1-T3; TD-M), "The dependency cycle is almost endless! You never know the ramifications of reusing a class" (P16-T1; Q-M). The primary problem was the developers' inability to determine the transitive costs associated with any dependency just by investigating the source code alone.

Using Gilligan it is easy to locate and navigate through source code dependencies. In contrast to the last category, here seven participants specifically commented about how Gilligan made it easy to navigate through the dependencies in the system. They commented that "It was easy to see what dependencies existed and navigate to them directly" (P15-T2; Q-G), and "[I liked] that I could just browse through the dependencies instead of using the editor, which would have taken forever" (P12-T3; TD-G). Using Gilligan's direct and transitive views, the participants would quickly scan for problematic dependencies: by considering the direct and indirect dependency counts they could quickly consider whether a dependency was worth investigating or would be inexpensive to reuse.

Manual investigations tend to be breath-first while Gilligan enables a more explorative methodology. Surprisingly, four participants specifically commented on how they explored their pragmatic reuse task in a more depth-first manner using Gilligan compared to a more breadth-first approach when investigating them manually; during the experiment we noted that many of the participants changed their behaviours in similar ways. "Manually it was really a breadth-first process whereas with Gilligan I could jump around more to consider what I wanted without losing that higher perspective. This is more natural because it allows you to follow your way throughout the code without stopping yourself in fear of getting lost or forgetting where you were" (P14). This comment sums up the difference Gilligan affords developers: exploring specific dependency chains is straightforward as it is easier to get back to the starting state; developers do not need drill down through many source code editors which may obscure from where they started their investigation. It is important to note that these strategies were not strict depth-first and breadth-first traversals of the dependency tree but had general tendencies consistent with these approaches. It could be considered that developers performed a best-first navigation strategy; what is interesting in this light is that the strategies were so divergent between the two treatments.

Gilligan helps developers manage the complex details that arise during pragmatic reuse tasks. Maintaining an accurate mental model of a pragmatic reuse task becomes more difficult as task size increases; however, the larger the reuse task, the greater the potential benefit of performing the task (in terms of productivity). By tracking decisions and supporting dependency investigation, Gilligan helped the participants feel more on top of their tasks; even in the case of the TD task, which was chosen specifically to be overwhelming: "Gilligan gave me a chance with [the third task] instead of rejecting [the task] right away. This way I know for sure that the task is bad instead of going by a gut feeling" (P2), and "For the [third task] I might have given up sooner but with Gilligan there is little risk in trying to reuse the code; it didn't matter if the task was huge because I could still keep track of what I was doing" (P10).

Gilligan encourages and assists developers in investigating alternative reuse strategies. When performing pragmatic reuse tasks manually, participants were reluctant to investigate alternative plan decisions, "In the manual case there was a lot of hesitiation to trying new things due to the level of commitment it would require" (P10). In contrast, 11 participants commented about how Gilligan encouraged them to variations on their reuse plan to find the best solution for their task: "[Using Gilligan] I can quickly see the results [of my decisions]; this is useful because it doesn't take 20 minutes to implement a single decision [as it might manually]" (P6-T1; RA-G). This had a positive impact on their task as they thought that, "Because I tried different plan alternatives [with Gilligan] I think my overall quality is better" (P14). Being able to rapidly prototype a decision using Gilligan and instantly see the ramifications of that decision on the reused source code helped participants avoid many of the pitfalls that befell the manual participants.

Most of the manual work required during a pragmatic reuse task is conceptually simple but labour intensive. 14 participants commented that manually performing the reuse task took longer than it should have. These comments all had similar themes, in that the steps they had to follow were not complicated, just tedious: "[The task] was difficult due to the robotic interactions; it was very repetitive" (P4-T1; Q-M). In contrast, 14 participants also noted that Gilligan greatly reduced the amount of manual effort they had to invest in a pragmatic reuse task, "Gilligan didn't force me to do anything I didn't want to do. Manually it was just more work, 2 orders of magnitude more" (P14-T2; Q-M).

9.7.3 Prompted concept categories

The three prompted concept categories are comprised of 16 themes and 66 sub-themes. The first two originated in questions that were asked of the participants, one before the study and one after. Before the participants started they were asked some background questions about pragmatic reuse tasks. After they had finished they were asked how they thought Gilligan could be further improved in the future. Finally, a large number of task-specific observations were made as the participants worked

through the experimental treatments.

The results for these three categories are reported more quantitatively than in the last section; the participants' answers were grouped and enumerated and are reported in more succinct form than for the organic concept categories. A complete description of each of these tables, and other prompted reuse categories can be found in Section G.2.

Answers to questions about pragmatic reuse. At the beginning of the experiment we asked each participant about their experience with pragmatic reuse tasks. During the card sort we distilled their responses down to the following tables.

Pragmatic Reuse Frequency	# of Participants
Frequently	6
Sometimes	7
Rarely	3

Table 9.5: Frequency that participants perform pragmatic reuse tasks (see Section G.2.1.1 for complete details).

Pragmatic Reuse Rationale	# of Participants
The code I need already exists.	9
It is faster than writing the code from scratch.	9
To use the existing code as an exemplar.	9
It is easier than writing the code from scratch.	6
To preserve existing encapsulation of the existing code.	3

Table 9.6: Participant's rationale for performing pragmatic reuse tasks (see Section G.2.1.2 for complete details).

Pragmatic Reuse Impediments	# of Participants
Uncertainty: Will it take a lot of work to reuse the code?	6
Propagating future changes can be difficult.	4
Keeping variable names consistent between systems.	3
Risk reusing badly-written code.	2
Might reuse source code you don't understand.	2

Table 9.7: Impediments to pragmatic reuse tasks identified by participants (see Section G.2.1.3 for complete details).

Suggested changes to Gilligan. After having participants performing 48 experimental trials and approximately 48 hours worth of development time using the third Gilligan prototype, we received a great deal of feedback about how the tool was effective and how it could be improved. These changes can be split into general feature requests (Table 9.8) and specific usability fixes (Table 9.9).

Suggested Gilligan Improvement	# of Participants
Increase navigation performance.	3
Provide an explicit indication of progress.	3
Link validation and enactment.	3
Link the editor with the structural views.	3
Support incremental building.	2
Add a dependency cost recommender.	1

Table 9.8: Suggested improvements for Gilligan (see Section G.2.2.1 for complete details).

Gilligan User Interface Shortcomings	# of Participants
Validation view is backwards.	5
Selection behaviour is awkward.	4
Views can be confusing.	4
Greyed-out text can hide details.	2

Table 9.9: Gilligan user interface shortcomings (see Section G.2.2.2 for complete details).

9.8 Discussion

9.8.1 Characterization of good tasks as bad ones

Both of the block B1 tasks were considered to be ideal candidates for pragmatic reuse. From Table 9.10 we can see that only one participant, performing the manual reuse case, thought the QIF parser was a poor reuse task. For the aTunes task 3 of 8 manual developers felt the reuse task was poor. In both cases we would disagree. These feelings lend evidence to the fact that developers manually performing a pragmatic reuse task are more pessimistic about it. We posit that developers manually performing a pragmatic reuse task are more likely to give up than those developers using Gilligan for the same task.

9.8.2 Giving up on bad tasks

In addition to the statistical findings that participants gave up on the third task more quickly using Gilligan, there is also qualitative data than further expands on this finding.

	Good Task	Bad Task
QIF Parser		
Manual	6	2
Gilligan	8	0
Total	*14*	*2*
Related Artists		
Manual	4	4
Gilligan	7	1
Total	*11*	*5*
Torrent Downloader		
Manual	2	6
Gilligan	1	7
Total	*3*	*13*

Table 9.10: # of participants who felt the task was good or bad.

Several developers made explicit statements about assessing tasks more confidently using Gilligan (see Section G.1.4.4 for more details). P14 said "I explored the task for a shorter duration [using Gilligan] but was more confident in my decision to surrender", while P5 stated, "I would be more confident predicting the eventual success or failure of a task [using Gilligan]."

One of our sub-themes identified from the qualitative data (see Section G.1.3.4 for full details) found that developers felt Gilligan gave them more of a chance, even on a bad pragmatic reuse task, than they would have had manually: "Gilligan gave me a chance with [the third task] instead of rejecting [the task] right away" (P2) and, "For the [third task] I might have given up sooner but with Giligan there is little risk in trying to reuse the code" (P4).

Anecdotally, we also mention that based on our timestamps that noted when a developer first vocalized that a task was a bad idea, the participants performing the Gilligan-supported tasks first stated that the task might be a bad idea early in the reuse task, generally within the first 10 minutes.

9.8.3 Gilligan improvements

Most of the recommended improvements our participants made for both Gilligan and its UI can be resolved with simple engineering work. Of all the recommended changes two of them stand out most prominently: better integration with the IDE and a recommendation system for pragmatic reuse plans. The integration aspect is interesting because our participants recommended several ways to better integrate Gilligan with the standard Eclipse tool suite that we would not have otherwise considered. They wanted the ability to navigate from the Eclipse problem view back into the Gilligan structural views; they also wanted to navigate from the editor to the abstraction in the structural views. These desires show great promise as they demonstrate how comfortable the participants were with the Gilligan environment; this integration would enable them to comfortably exist within the

Gilligan perspective without any specific external tool support.

The recommendation system is also interesting because we can definitely see how a recommender could keep a developer from pursuing ill-advised paths, or to make it clearer to them when dependencies were very cheap and could be reused without penalty.

9.8.4 Limitations

This experiment had two main limitations. Because block B1 comprised only two tasks it is difficult for us to generalize our results to the wider arena of pragmatic reuse tasks. While we believe that there was nothing special about these two tasks, further evaluation on a wider variety of tasks would be needed to gain confidence in the generalizability of these results.

Similarly, for block B2, we only had one task. This certainly is not enough to generalize the findings but provides initial evidence that is promising.

While our study consisted of only 16 participants, we believe that they were from a heterogeneous sample that would generalize at least to Eclipse-using Java developers. This study could have involved more participants, but considering the time required of the participants for this experiment it was deemed excessively taxing to further widen the study as gaining access to industrial developers, even for three hours at a time, is difficult.

9.9 Summary

In this chapter we have described a holistic evaluation of the Gilligan tool suite. This evaluation builds on all of the findings from our previous investigations (Chapter 4, Section 6.3, Section 7.3, and Section 8.3) by having 16 participants perform 3 complete pragmatic reuse tasks each. These tasks involved both planning and performing pragmatic reuse tasks and the tasks were considered successful if the reused code was able to be executed by a test harness.

Based on this evaluation we make these quantitative findings:

- We accept our hypothesis H-1: through an ANOVA analysis, we reject the null hypothesis relative to H-1, meaning that participants using Gilligan were able to complete their tasks in significantly less time than developers performing the same tasks with standard IDE tools.

- We accept our hypothesis H-2: Fisher's test rejects the null hypothesis relative to H-2, meaning that participants using Gilligan were significantly more likely to successfully complete their pragmatic reuse tasks (16 of 16 treatments were successful for Gilligan trials while only 11 of 16 were successful for the manual trials).

- We accept our hypothesis H-3: the MWW test rejects the null hypothesis relative to H-3, meaning that participants spent significantly less time investigating an infeasible reuse task than those investigating the same task manually.

The evaluation presented in this chapter did not only focus on quantitative approaches but also leveraged qualitative techniques through the large volume of participant statements we collected over the 48 hours we spent with them while performing this experiment. Using a grounded-theory approach we found five key conceptual categories that were on developers' minds as they were performing these tasks. While some of them, such as program understanding and the mechanics of performing a pragmatic reuse task are not all that surprising, the importance of the other three—dependency identification, mental models, and hypothesis testing— were surprising.

Two groups of statements in particular, from the qualitative side of this experiment, jumped out at us: "Gilligan didn't force me to do anything I didn't want to do. Manually it was just more work, 2 orders of magnitude more", and "I explored the task for a shorter duration [using Gilligan] but was more confident in my decision to surrender. Manually would have taken longer but I would be less confident in my decision to give up." These statements represent sentiments that many of our participants had in this experiment: Gilligan can help one understand and to perform these tasks in ways that are much more effective than standard development practice.

Chapter 10

Discussion

10.1 Alternative reuse strategies

This dissertation has mainly presented reuse strategies involving the reuse of a feature by carefully building a reuse plan from some small fragment (usually a method) of source code. While Gilligan is well-suited to this type of reuse strategy, there are two-other approaches that developers may want to employ:

- *Top-down reuse.* Rather than build up from a small kernel of knowledge, a developer could choose to accept large amounts of functionality and then to trim down the parts they do not need to reuse. In this way they would be selectively cutting out functionality, rather than selectively including it.

- *Hybrid reuse.* Alternatively to carefully adding or removing functionality, a developer could choose to build up the code they want to reuse by balancing the necessity of the functionality against the ease of reuse. That is, the developer would not worry about reusing limited amounts of extraneous functionality if it were cheap to reuse. This approach can make it easier to accomplish a reuse task, especially to evaluate the effectiveness of a prototype and to ensure that the code actually does what is desired before a more considered approach is taken. We have found this approach to be effective in practice, although care must be taken to ensure that too much extraneous functionality is not reused.

While we have not discussed Gilligan in terms of these alternative strategies within this dissertation, both strategies can be easily employed by a developer using Gilligan.

A fourth approach, involving taking a dependency on a whole system, is not complementary to these other three approaches and generally not an acceptable reuse technique in practice. Reusing an entire system generally leads to the phenomenon known as bloat in which the footprint of a software system grows enormously with unneeded functionality; in addition to adding a large body of unused

code to the system, this approach also complicates comprehension activities as it can be difficult to identify which portions of the reused system must be investigated to fix a defect or add a new feature. Instead, we can see active attempts in industry to strip away unnecessary functionality, for example, in the Eclipse Rich Client Platform [Edgar, 2003a,b].

10.2 Limitations of approach

There are two main limitations of our approach to pragmatic software reuse:

1. *Planning overhead can overwhelm performance benefits for small reuse tasks.* A developer must expend a certain amount of effort to create a pragmatic reuse plan; when the effort required exceeds the amount of time the developer would need to perform the reuse task manually, our approach does not make sense. As such, performing pragmatic reuse tasks using our approach for small tasks is not recommended. As a guideline, we have found that tasks that only reuse a self-contained set of a handful of methods or less are generally easier to perform by hand. Interestingly, we expected that the QIF Parser task outlined in Chapter 9 would be easier to perform manually due to the fact that the functionality was well contained; however, even in this case, developers were much quicker using Gilligan.

2. *Non-code artifacts are not analyzed and cannot be reused.* Our approach to planning pragmatic reuse tasks only considers the static structure of the code. Because of this, Gilligan cannot help developers reuse documentation, configuration settings, data files, or any other infrastructure associated with a project. While 'design-level' features can be reused using Gilligan, our intent was not to explicitly enable the reuse of designs themselves, but of the code that embodies those designs.

10.3 Evaluation

We conducted many surveys, case studies, and controlled experiments while investigating how developers perform pragmatic reuse tasks using our Gilligan prototypes. Industrial developers were involved at each significant phase of these evaluations and we relied heavily on their opinions and insights to guide our research. While none of these studies were excessively large, we are confident that our results truly demonstrate that our approach improves the state-of-the-art for developers performing pragmatic reuse tasks. The breadth of tasks and developers who have participated in our experiments and case studies has given us confidence that our approaches can be effective in industrial settings.

As mentioned in Section 6.4, we encountered several organizational obstacles when getting industrial developers to perform pragmatic reuse tasks on their own code. These problems mainly arose

from concerns raised by our participant's managers and their company's legal departments with respect to releasing important internal data to us. As we needed this data to understand both the nature of the tasks they were performing, as well as the process by which they undertook them, we opted in later studies to get developers to partake in controlled experiments rather than case studies to avoid such problems. The consequence of this choice was increased internal validity as we could much more effectively compare how these industrial developers performed relative to one and other with various treatments, but at the expense of decreased external validity as the developers were performing far fewer and more constrained pragmatic reuse tasks.

One issue that commonly arises in evaluations of developer productivity is that of the extreme range of abilities of developers. While this range of abilities certainly exists, we never encountered a situation in any of our experiments where we considered it to be a problem. In each experiment, developers performed two or more tasks using various treatments and the situation where a single developer failed at all their assigned tasks never occurred.

10.4 Future work

Based on all our observations and conversations with developers throughout this research process, we have identified three design requirements that a fourth prototype of Gilligan should consider.

DG 4.1: *Support identifying sub-type relationships more effectively.* Currently, Gilligan shows parent classes and interfaces as direct dependencies of a class; however, for a calling relationship where a method may be declared within a subclass, this support is insufficient. In this case, Gilligan simply shows the call relationship to the statically-derivable method in the super type. Gilligan should perform additional analysis to help developers avoid being trapped by complexities that may be obscured by the type hierarchy.

DG 4.2: *Adaptively update direct and indirect dependency counts.* The direct and indirect dependency counts provided within Gilligan give the developer an indication of how many individual dependencies one structural element has. These counts are currently static, but should not be; if a developer rejects an expensive dependency, the count for that element (and all other elements depending on the newly-rejected element) should be dynamically updated to reflect this change.

DG 4.3: *Provide an active recommendation system to help developers avoid expensive structural elements.* Gilligan currently acts as a mechanism to help developers plan and perform pragmatic reuse tasks. While the tool can help them to be more systematic and guide them to dependencies they should investigate, it does not currently explicitly help them avoid those dependencies that could be expensive to reuse. Gilligan should incorporate a recommendation system that considers not only the structural relationships within the

system, but also all the decisions the developer has already made in their reuse plan; as the developer makes further decisions, the recommendations can be refined to better match the developer's context.

There are also several other directions we would like to consider for the Gilligan tool as well as pragmatic reuse research in general:

- *Create reusable components using Gilligan.* Gilligan aims to help developers extract some functional unit that was not necessarily designed in a reusable fashion from an existing system and integrate it into their own project. Gilligan could be adapted to take a reuse plan and extract the code into a standalone component, instead of integrating it into the developer's project. The developer could then use the component in a black-box manner, thus enabling other developers to reuse the same component in the future.

- *Use Gilligan to compose new systems.* We have advocated pragmatic reuse as a mechanism to reuse code within existing systems; however, we have not investigated using pragmatic reuse to reuse code within *empty* systems. While Gilligan does not impose any restrictions on the target project that would limit this approach, additional support would enable the source code from multiple pragmatic reuse tasks to be integrated more effectively.

- *Support reusing and adapting existing test suites applicable for reused code.* While we would argue that reusing tested, proven source code is generally better than writing new code from scratch, this belief would be further reinforced if test cases associated with reused source code could be detected and reused as well.

- *Provide feedback to original source code authors when their code is reused pragmatically.* One of the main reasons developers perform pragmatic reuse tasks is that they lack the ability to change the original code; that said, providing feedback to the owner of the code about the pragmatic reuse task could both help them understand how their code is being used in reality and give them data to consider future changes to the code to make it more reusable by others.

- *Enable changes to the original code to be applied to the reused code.* The primary negative property associated with code clones is that having clones increases maintenance effort as any change must be made many times to propagate throughout a system. In the pragmatic reuse scenario this is even more complex as the reused code may be spread between many projects, some of which the original developer may not even know about. While developers who have reused code may explicitly *not* want to have their code updated (Cordy [2003] mentions this explicitly), the ability to do so could still be supported for those who want it; by tracking exactly where Gilligan got the source code that was reused, as well as where the reused code was placed, Gilligan could be augmented to guide a developer through the application of changes that were made to the original code to the relevant locations within the reused code.

Chapter 11

Conclusion

Reusing software artifacts has been demonstrated to be an effective mechanism to decrease development times, software product costs, and defect rates. While black-box reuse is most often promoted as the "way" reuse should be effected in an organization, practice shows that a large proportion of reused software artifacts are reused in a pragmatic way from source code that was not designed for reuse.

Unfortunately, pragmatic reuse tasks are often stigmatized as the "wrong" way to reuse code; this stigma has developed over time due to the non-systematic, ad hoc nature of these tasks that can lead to their lack of proper consideration and performance.

The thesis of this dissertation is that by providing developers with mechanisms to plan pragmatic reuse tasks and capture their intent in a structured way, we can enable developers to perform pragmatic reuse tasks in less time and with greater confidence. We conducted five evaluations to accumulate evidence in support of our thesis.

Through a survey of 12 industrial developers we found that these developers do perform pragmatic reuse tasks. An uncontrolled industrial case study later found that four industrial developers were able to create pragmatic reuse plans with the first prototype of our pragmatic reuse tool, called Gilligan.

A controlled experiment found that developers frequently make errors locating structural dependencies in source code manually, finding only 45% of structural dependencies from an initial seed while developers using our second Gilligan prototype were able to find 87% of the correct dependencies.

Through a case study evaluating our third prototype of Gilligan we found that through automating the enactment of a pragmatic reuse plan we could greatly reduce the number of compilation errors a developer must fix while enacting a pragmatic reuse plan by more than 65% and decrease the number of cognitive decisions they must consider by more than 84%. Through another controlled experiment we found that developers enacting pragmatic reuse plans were significantly more likely to succeed (8 of 8 cases with Gilligan compared to 4 of 8 cases without) and in much less time (between

60% and 75% less time) using Gilligan.

Finally, we performed a large-scale controlled experiment that found a statistically significant time savings (38%) for developers using Gilligan to perform a holistic pragmatic reuse task involving both planning and performing the reuse task. Through qualitative observations made during this experiment we also identified five conceptual areas that are of keen concern to developers performing pragmatic reuse tasks.

By employing our model of pragmatic reuse through our tool suite, we have made pragmatic reuse tasks less ad hoc and more systematic. This has increased the likelihood of a developer successfully completing a pragmatic reuse task. This increased likelihood translates into the potential for greater adoption of pragmatic reuse tasks within industrial projects which could lead to better productivity, lower costs, and lower rates of defects over software written from scratch.

11.1 Contributions

This dissertation has made five main contributions:

1. A model for capturing developer intent while planning a pragmatic reuse task (the pragmatic reuse plan).

2. The Gilligan prototype tool that provides a visual representation of a pragmatic reuse plan and a mechanism to help developers explore and triage the dependencies within the source code they are investigating for reuse as well as semi-automatically transforming the reused code according to the reuse plan.

3. Evidence that developers can much more accurately identify structural dependencies using Gilligan than standard tools.

4. Evidence that Gilligan can make a statistically-significant impact on the amount of time required to perform pragmatic reuse tasks.

5. A categorization of the key cognitive aspects facing developers as they perform pragmatic reuse tasks.

Bibliography

Jean-Raymond Abrial (1996). *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press → p. 37

Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer (1980). Specification language. In R. M. McKeag and A. M. Macnaughten (Eds.), *On the Construction of Programs*, 343–410. Cambridge University Press → p. 37

Samuel A. Ajila and Di Wu (2007). Empirical study of the effects of open source adoption on software development economics. *Journal of Systems and Software* 80(9):1517–1529. doi:10.1016/j.jss.2007.01.011 → p. 1

Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo (1996). How reuse influences productivity in object-oriented systems. *Communications of the ACM* 39(10):104–116. doi:10.1145/236156.236184 → p. 1

Paul G. Bassett (1997). The theory and practice of adaptive reuse. *SIGSOFT Software Engineering Notes* 22(3):2–9. doi:10.1145/258368.258371 → p. 19

Ira D. Baxter (2002). DMS: Program transformations for practical scalable software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, 48–51. doi:10.1145/512035.512047 → p. 22

William Berg, Marshall Cline, and Mike Girou (1995). Lessons learned from the os/400 oo project. *Communications of the ACM* 38(10):54–64. doi:10.1145/226239.226253 → p. 16

T. J. Biggerstaff (1994). The library scaling problem and the limits of concrete component reuse. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 102–109. doi:10.1109/ICSR.1994.365806 → p. 1

Barry Boehm (1999). Managing software productivity and reuse. *Computer* 32(9):111–113. doi:10.1109/2.789755 → p. 1

Fred P. Brooks, Jr. (1987). No silver bullet: Essence and accidents of software engineering. *Computer* 20(4):10–19. doi:10.1109/MC.1987.1663532 → p. 1

Gianluigi Caldiera and Victor R. Basili (1991). Identifying and qualifying reusable software components. *Computer* 24(2):61–70. doi:10.1109/2.67210 → p. 19

Kunrong Chen and Václav Rajlich (2000). Case study of feature location using dependence graphs. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, 241–247. doi:10.1109/WPC.2000.852498 → p. 21

Y.-F. R. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach (1995). Ciao: A graphical navigator for software and document repositories. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 66–75. doi:10.1109/ICSM.1995.526528 → p. 21

Juliet M. Corbin and Anselm Strauss (1990). Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology* 13(1):3–21. doi:10.1007/BF00988593 → p. 107, 195

James R. Cordy (2003). Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, 196–205. doi:10.1109/WPC.2003.1199203 → p. 1, 21, 123

James R. Cordy (2006). The TXL source transformation language. *Science of Compuer Programming* 61(3):190–210. doi:10.1016/j.scico.2006.04.002 → p. 22

Rylan Cottrell (2008). *Semi-automating Small-Scale Source Code Reuse via Structural Correspondence*. Master's thesis, University of Calgary → p. 20

Rylan Cottrell, Robert J. Walker, and Jörg Denzinger (2008a). Jigsaw: A tool for the small-scale reuse of source code. In *Companion of the International Conference on Software Engineering (ICSE)*, 933–934. doi:10.1145/1370175.1370194 → p. 20

Rylan Cottrell, Robert J. Walker, and Jörg Denzinger (2008b). Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 214–225. doi:10.1145/1453101.1453130 → p. 20

Brian de Alwis and Gail C. Murphy (2006). Using visual momentum to explain disorientation in the Eclipse IDE. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC)*, 51–54. doi:10.1109/VLHCC.2006.49 → p. 24

Edsger W. Dijkstra (1972). Notes on structured programming. In *Structured Programming*, chap. 1, 1–82. London, UK: Academic Press Ltd. → p. 1, 19

Thomas Drake (1996). Measuring software quality: A case study. *Computer* 29(11):78–87. doi:10.1109/2.544241 → p. 17

Ekwa Duala-Ekoko and Martin P. Robillard (2007). Tracking code clones in evolving software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 158–167. doi:10.1109/ICSE.2007.90 → p. 21

Nick Edgar (2003a). Bug 36967: Enable Eclipse to be used as a rich client platform. https://bugs.eclipse.org/bugs/show_bug.cgi?id=36967. Last checked December 2008. → p. 121

Nick Edgar (2003b). Eclipse Rich Client Platform UI. http://www.eclipse.org/rcp/generic_workbench_summary.html. Last checked December 2008. → p. 121

Thomas Eisenbarth, Rainer Koschke, and Daniel Simon (2003). Locating features in source sode. *IEEE Transactions on Software Engineering* 29(3):210–224. doi:10.1109/TSE.2003.1183929 → p. 22

Jacky Estublier and German Vega (2005). Reuse and variability in large software applications. *SIGSOFT Software Engineering Notes* 30(5):316–325. doi:10.1145/1095430.1081757 → p. 19

Martin S. Feather (1989). Reuse in the context of a transformation-based methodology. In Ted J. Biggerstaff and Alan J. Perlis (Eds.), *Software Reusability*, vol. 1: Concepts and Models, chap. 14, 337–359. Addison–Wesley → p. 22

G. Fischer (1987). Cognitive view of reuse and redesign. *IEEE Software* 4(4):60–72. doi:10.1109/MS.1987.231065 → p. 23

William B. Frakes and Christopher J. Fox (1995). Sixteen questions about software reuse. *Communications of the ACM* 38(6):75–88. doi:10.1145/203241.203260 → p. 2

William B. Frakes and Kyo Kang (2005). Software reuse research: Status and future. *IEEE Transactions on Software Engineering* 31(7):529–536. doi:10.1109/TSE.2005.85 → p. 2, 23

William B. Frakes and Giancarlo Succi (2001). An industrial study of reuse, quality, and productivity. *Journal of Systems and Software* 57(2):99–106. doi:10.1016/S0164-1212(00)00121-7 → p. 1

John E. Gaffney, Jr. and R. D. Cruickshank (1992). A general economics model of software reuse. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 327–337. doi:10.1145/143062.143150 → p. 1

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley → p. 22

David Garlan, Robert Allen, and John Ockerbloom (1995). Architectural mismatch: Why reuse is so hard. *IEEE Software* 12(6):17–26. doi:10.1109/52.469757 → p. 1, 3, 6, 85

E. S. Garnett and J. A. Mariani (1990). Software reclamation. *Software Engineering Journal* 5(3):185–191 → p. 2

Mohamed G. Gouda and Ted Herman (1991). Adaptive programming. *IEEE Transactions on Software Engineering* 17(9):911–921. doi:10.1109/32.92911 → p. 22

William G. Griswold and David Notkin (1993). Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology* 2(3):228–269. doi:10.1145/152388.152389 → p. 23

Brian Henderson-Sellers (1996). *Object-Oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. → p. 15

Johannes Henkel and Amer Diwan (2005). CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 274–283. doi:10.1145/1062455.1062512 → p. 23

Reid Holmes and Gail C. Murphy (2005). Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 117–125. doi:10.1145/1062455.1062491 → p. 20

Reid Holmes and Robert J. Walker (2007a). Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 447–457. doi:10.1109/ICSE.2007.83 → p. 25, 42

Reid Holmes and Robert J. Walker (2007b). Task-specific source code dependency investigation. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 100–107. doi:10.1109/VISSOF.2007.4290707 → p. 56

Reid Holmes and Robert J. Walker (2008). Lightweight, semi-automated enactment of pragmatic-reuse plans. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 330–342. doi:10.1007/978-3-540-68073-4_35 → p. 73

Reid Holmes, Robert J. Walker, and Gail C. Murphy (2006). Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering* 32(12):952–970. doi:10.1109/TSE.2006.117 → p. 20, 38

Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto (2003). Component Rank: Relative significance rank for software component search. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 14–24. doi:10.1109/ICSE.2003.1201184 → p. 19

Patricia Jablonski and Daqing Hou (2007). CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the Workshop on Eclipse Technology eXchange (eTX)*, 16–20. doi:10.1145/1328279.1328283 → p. 20

Ralph E. Johnson and Brian Foote (1988). Designing reuseable [*sic*] classes. *Journal of Object-Oriented Programming* 1(2):22–35 → p. 19

Cory Kapser and Michael W. Godfrey (2006). "Cloning considered harmful" considered harmful. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 19–28. doi:10.1109/WCRE.2006.1 → p. 21

Gregor Kiczales, , Jim des Rivières, and Daniel Bobrow (1991). *The Art of the Metaobject Protocol*. MIT Press → p. 22

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold (2001). An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 327–353. doi:10.1007/3-540-45337-7 → p. 22

Miryung Kim and David Notkin (2006). Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, 58–64. doi:10.1145/1137983.1137999 → p. 21

Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy (2005). An empirical study of code clone genealogies. In *Proceedings of the Joint European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 187–196. doi:10.1145/1081706.1081737 → p. 19, 21

Charles W. Krueger (1992). Software reuse. *ACM Computing Surveys* 24(2):131–183. doi:10.1145/130844.130856 → p. 1, 2, 3

Charles W. Krueger (2000). Software product line reuse in practice. In *Proceedings of the Symposium on Application-Specific Systems and Software Engineering Technology (ASSET)*, 117. doi:10.1109/ASSET.2000.888062 → p. 1, 19

B. M. Lange and T. G. Moher (1989). Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, 69–73. doi:10.1145/67449.67465 → p. 2

Filippo Lanubile and Giuseppe Visaggio (1997). Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering* 23(4):246–259. doi:10.1109/32.588543 → p. 19

J. L. Lawrence (1985). Why is software always late? *SIGSOFT Software Engineering Notes* 10(1):19–30. doi:10.1145/1012443.1012445 → p. 16

Bongshin Lee, Cynthia S. Parr, Catherine Plaisant, Benjamin B. Bederson, Vladislav D. Veksler, Wayne D. Gray, and Christopher Kotfila (2006). TreePlus: Interactive exploration of networks with enhanced tree layouts. *IEEE Transactions on Visualization and Computer Graphics* 12(6):1414–1426. doi:10.1109/TVCG.2006.106 → p. 23

Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher (2007). CodeGenie: A tool for test-driven source code search. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 917–918. doi:10.1145/1297846.1297944 → p. 19

Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao (1994). Adaptive object-oriented programming using graph-based customization. *Communications of the ACM* 37(5):94–101. doi:10.1145/175290.175303 → p. 22

Steve McConnell (2004). *Code Complete*. Redmond, WA, USA: Microsoft Press, 2nd ed. → p. 17

D. Mcilroy (1968). Mass-produced software components. In *Software Engineering: Report on a Conference by the NATO Science Committee*, 138–155 → p. 1, 18, 19

Bertrand Meyer (1999). On to components. *Computer* 32(1):139–140. doi:10.1109/2.738312 → p. 1, 19

Mira Mezini and Klaus Ostermann (2002). Integrating independent components with on-demand remodularization. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 52–67. doi:10.1145/582419.582426 → p. 19

Matthew B. Miles and Michael Huberman (1994). *Qualitative Data Analysis: An Expanded Sourcebook (2nd Edition)*. Thousand Oaks, CA.: Sage Publications, Inc. → p. 107, 195

M. Morisio, M. Ezran, and C. Tully (2002). Success and failure factors in software reuse. *IEEE Transactions on Software Engineering* 28(4):340–357. doi:10.1109/TSE.2002.995420 → p. 2

H. A. Müller and K. Klashinsky (1988). Rigi: A system for programming-in-the-large. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 80–86 → p. 21, 23

J. M. Neighbors (1989). Draco: A method for engineering reusable software systems. In Ted J. Biggerstaff and Alan J. Perlis (Eds.), *Software Reusability*, vol. 1: Concepts and Models, 295–319. New York, NY, USA: Addison–Wesley. doi:10.1145/73103.73115 → p. 19

William F. Opdyke (1992). *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois at Urbana-Champaign → p. 23

D.L. Parnas (1976). On the design and development of program families. *IEEE Transactions on Software Engineering* 2(1):1–9. doi:10.1109/TSE.1976.233797 → p. 1, 19

Jeffrey Parsons and Chad Saunders (2004). Cognitive heuristics in software engineering: Applying and extending anchoring and adjustment to artifact reuse. *IEEE Transactions on Software Engineering* 30(12):873–888. doi:10.1109/TSE.2004.94 → p. 20, 24

Shari Lawrence Pfleeger (1994). Design and analysis in software engineering—Part 1: The language of case studies and formal experiments. *SIGSOFT Software Engineering Notes* 19(4):16–20. doi:10.1145/190679.190680 → p. 95

Jeffrey S. Poulin, Joseph M. Caruso, and Debera R. Hancock (1993). The business case for software reuse. *IBM Systems Journal* 32(4):567–594 → p. 1, 18

Lutz Prechelt (2000). An empirical comparison of seven programming languages. *Computer* 33(10):23–29. doi:10.1109/2.876288 → p. 17

Rubén Prieto-Díaz (1993). Status report: Software reusability. *IEEE Software* 10(3):61–66. doi:10.1109/52.210605 → p. 1, 2

Thiagarajan Ravichandran and Marcus A. Rothenberger (2003). Software reuse strategies and component markets. *Communications of the ACM* 46(8):109–114. doi:10.1145/859670.859678 → p. 1, 2, 19

S. P. Reiss (2005). The paradox of software visualization. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 19. doi:10.1109/VISSOF.2005.1684306 → p. 23, 58

Martin P. Robillard (2003). *Representing Concerns in Source Code*. Ph.D. thesis, University of British Columbia → p. 38

Martin P. Robillard (2005). Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 11–20. doi:10.1145/1081706.1081711 → p. 22

Martin P. Robillard and Gail C. Murphy (2002). Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 406–416. doi:10.1145/581339.581390 → p. 22

Martin P. Robillard and Gail C. Murphy (2007). Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology* 16(1):3. doi:10.1145/1189748.1189751 → p. 38

Mary Beth Rosson and John M. Carroll (1993). Active programming strategies in reuse. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 4–20. doi:10.1007/3-540-47910-4_2 → p. 2

Mary Beth Rosson and John M. Carroll (1996). The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction* 3(3):219–253. doi:10.1145/234526.234530 → p. 5

Marcus A. Rothenberger, Kevin J. Dooley, Uday R. Kulkarni, and Nader Nada (2003). Strategies for software reuse: A principal component analysis of reuse practices. *IEEE Transactions on Software Engineering* 29(9):825–837. doi:10.1109/TSE.2003.1232287 → p. 1

Chris Sadler and Barbara Ann Kitchenham (1996). Evaluating software engineering methods and tools—Part 4: The influence of human factors. *SIGSOFT Software Engineering Notes* 21(5):11–13. doi:10.1145/235969.235972 → p. 102

Thorsten Schafer, Michael Eichberg, Michael Haupt, and Mira Mezini (2006). The SEXTANT software exploration tool. *IEEE Transactions on Software Engineering* 32(9):753–768. doi:10.1109/TSE.2006.94 → p. 23

Richard W. Selby (2005). Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering* 31(6):495–510. doi:10.1109/TSE.2005.69 → p. 2, 19, 20, 23

Arun Sen (1997). The role of opportunism in the software design reuse process. *IEEE Transactions on Software Engineering* 23(7):418–436. doi:10.1109/32.605760 → p. 2, 24

Susan Elliott Sim and Margaret-Anne D. Storey (2000). A structured demonstration of program comprehension tools. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 184–194. doi:10.1109/WCRE.2000.891465 → p. 24

Janice Singer (1998). Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 139. doi:10.1109/ICSM.1998.738502 → p. 46

Thomas A. Standish (1984). An essay on software reuse. *IEEE Transactions on Software Engineering* 10(5):494–497 → p. 1

Neal Stephenson (1992). *Snow Crash*. Bantam Books → p. 2

M.-A. D. Storey, K. Wong, and H. A. Müller (2000). How do program understanding tools affect how programmers understand programs? *Science of Compuer Programming* 36(2-3):183–207. doi:10.1016/S0167-6423(99)00036-2 → p. 23

Giancarlo Succi, Luigi Benedicenti, and Tullio Vernazza (2001). Analysis of the effects of software reuse on customer satisfaction in an RPG environment. *IEEE Transactions on Software Engineering* 27(5):473–479. doi:10.1109/32.922717 → p. 1

Clemens Szyperski (2002). *Component software: Beyond object-oriented programming*. New York, NY, USA: ACM Press → p. 1, 19

John R. Taylor (1996). *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*. University Science Books, 2nd ed. → p. 105

Suresh Thummalapenta and Tao Xie (2007). Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 204–213. doi:10.1145/1321631.1321663 → p. 20

Michael Toomim, Andrew Begel, and Susan L. Graham (2004). Managing duplicated code with linked editing. In *Proceedings of the Symposium on Visual Languages and Human Centric Computing (VLHCC)*, 173–180. doi:10.1109/VLHCC.2004.35 → p. 20

Will Tracz (1990). Where does reuse start? *SIGSOFT Software Engineering Notes* 15(2):42–46. doi:10.1145/382296.382702 → p. 1

Jilles van Gurp and Jan Bosch (2002). Design erosion: Problems and causes. *Journal of Systems and Software* 61(2):105–119. doi:10.1016/S0164-1212(01)00152-2 → p. 1

Bruce W. Weide, William F. Ogden, and Stuart H. Zweben (1991). *Reusable software components*, vol. 33, 1–65. San Diego, CA, USA: Academic Press Professional, Inc. → p. 19

Daniel M. Yellin and Robert E. Strom (1997). Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems* 19(2):292–333. doi:10.1145/244795.244801 → p. 22

Appendix A

Industrial Pragmatic Reuse Survey

This chapter includes the supporting materials from the industrial survey performed at the outset of the research contained in this dissertation. The intent and findings of this survey is described in Chapter 4. We include the survey provided to the industrial developers (p. 135). The complete results of the Likert Scale questions are given in Table A.1 while the aggregate results are provided in Table A.2.

The supporting materials for the case studies discussed in Section 6.3 are provided in Section A.2 (p. 146). The industrial developers involved in this study were asked to return one copy of the second page of this questionnaire for each task they attempted.

Software Reuse Questionnaire

Please email responses to rtholmes@cs.ucalgary.ca

Thank you for filling out this questionnaire. This is the first part of a study I am conducting that examines whether developers reuse source code, and if so, how they do it, and how the reuse experience can be improved. Your feedback is critical to ensuring that this research is relevant to how industrial developers reuse source code.

Just to be clear, the definition I'm using for reuse may a little different than you're used to. This is what I mean by both reuse and the reuse process:

Use/Reuse: The distinction between use and reuse is somewhat subtle. Calling a method within a framework or reusable library is not reuse, it is simply using it. Reuse involves taking code and employing it in a new context. This definition of reuse does not make any statement about size: I am interested in the reuse of any code, from single lines and functions to full classes, packages, and features.

Examples of reuse:

- Extracting and integrating the code from Azureus that implements graph drawing functionality you want
- Copying and modifying a class from another project in your company that implements an AST visitor that performs a similar operation you want to support.
- Extracting a wizard from Eclipse that creates new projects and modifying it to create new templates in your system.

Reuse Process: By this definition of reuse the reuse process entails four primary steps:

1) Locating source code to be reused
2) Identifying which portions of the code are relevant to the task and make the decision to proceed (or not) with reusing the code
3) Extracting the code from the context in which it was developed
4) Integrating the code into a new development context

This questionnaire will be an important part of my investigation into the second step of the reuse process. Gilligan, the tool I am developing, will eventually address steps 2-4 in this process. The current prototype of Gilligan (the one I will get you to use) is only posed to address step 2.

Only I will have access to the raw results from this survey. I will process the results so that they cannot be attributed to individual developers / organizations. I will not quote you without your permission.

The goal of this questionnaire is to identify some background information about your development practices, determine whether you have reused source code in the past, and determine the nature of the code you have reused (if you have).

I will happily call you at anytime to conduct this questionnaire on the phone, or to follow up on the answers you gave me, if it is more convenient for you. Due to the time-sensitive nature of this work (paper deadline) I would appreciate your responses as soon as possible. When answering the multiple choice questions please mark your selection in bold. For the free form answers, feel free to delete the underlined sections and fill in your response. Please use as much space as you need.

Your responses are very important to me as I will use them to guide the rest of my research. I will incorporate your feedback into my tool as much as possible. While my tool will remain a research prototype, you are entitled to any future versions (if you want them). Furthermore, if you know any other developers (in your organization or not) who you think would be interested in answering this questionnaire (or performing the full evaluation) please let me know; I am always looking for new participants.

I am also preparing a tool demonstration so you can test out the Gilligan system (as it exists at least) and give me feedback. If you are interested in trying the demo, please let me know. This will involve a short demo and then you will be able to use the tool to explore any existing code you would be interested in reusing. If, while you are working, you could keep an eye out for code you would be interested in reusing (or are reusing) write them down so you can test my tool on these cases when I get it to you.

Thanks again for your help. I really look forward to reading your responses and am excited to see how the tool works for you.

--Reid Holmes

(rtholmes@cs.ucalgary.ca)

Background Questionnaire

Please answer based on your experiences in industry. This does not need to be constrained to your current job, project, or the tasks you are working on.

1. Name: ______________________
2. Job Title: ____________________
3. Company: ___________________
4. Do you write code on a daily basis? Yes No
5. How long have you been writing code in industry? ______ months/years
6. Have you ever reused code from another project? Yes No

   Note: The definition of reuse I am using means copying any amount code from the location in which it was developed into a new context. Even reusing code within a project is reuse itself. (see the page 1 para. 3 for more detail)

   If no, please skip questions 6 a-g and continue to question 7.

   a. When you reuse code, why do you do it?

   ____

   ____

   ____

   ____

   ____

   b. How often do your development tasks involve source code reuse in any form? Does the task you are working on influence whether or not you reuse code?

   ____

   ____

   ____

   ____

   ____

c. Generally, how large are these reuse tasks (in terms of lines of code, # of methods / classes / packages, etc.)? What is the smallest / largest they have been?

____

____

____

____

____

d. Do you use any tools to complete these reuse tasks? What services do these tools provide that help you reuse code?

____

____

____

____

____

e. Are these tools sufficient for completing these tasks? How could they be improved? What support do you need from a tool when you are planning / executing a reuse task?

____

____

____

____

____

____

f. What factors do you consider when you decide whether to reuse a piece of source code or re-implement it from scratch?

____

____

____

____

____

____

g. Are there any specific questions do you want answered about the code you are considering reusing? What challenges / questions do you face when you are trying to make the decision from (f) for a particular piece of code?

____

____

____

____

____

7. Do you consider reusing source code features to be good or bad practice? Specifically, do you consider copying portions of code from existing systems to use within new ones to be advantageous or disadvantageous?

____

____

____

____

____

____

8. Do you use any form of abstraction to keep track of the relevant details and decisions you have made about a feature while you are navigating its source? Is having a coherent mental picture of a piece of software important for successfully completing a task? Do you use any specific tools or techniques (notes, pictures, annotations in the code, etc.) to codify this model?

   ____

   ____

   ____

   ____

   ____

   ____

The next three tables include specific questions about development in general, IDE's (in particular Eclipse), and finally reuse itself. Please answer based on your experiences in industry. Feel free to insert comments below each table if you have any specific comments you would like to make about a particular question.

Development Questions:	strongly disagree	disagree	somewhat disagree	no opinion	somewhat agree	agree	strongly agree
I primarily develop new features.	1	2	3	4	5	6	7
I primarily maintain existing features.	1	2	3	4	5	6	7
I have flexibility in choosing how to complete my development tasks.	1	2	3	4	5	6	7
Creating reusable software is encouraged in my organization.	1	2	3	4	5	6	7
Reusing software is encouraged in my organization.	1	2	3	4	5	6	7
My organization has a large amount of code available to be reused.	1	2	3	4	5	6	7
Portions of features I am developing already exist in other software systems (and I have access to the source code) .	1	2	3	4	5	6	7

Comments:

Environment Questions:	strongly disagree	disagree	somewhat disagree	no opinion	somewhat agree	agree	strongly agree
I use an IDE (while working with code).	1	2	3	4	5	6	7
I use advanced IDE tools (e.g. refactoring support, type hierarchy navigation).	1	2	3	4	5	6	7
My primary development environment is Eclipse.	1	2	3	4	5	6	7
My primary development language is Java.	1	2	3	4	5	6	7
I use program understanding tools (beyond those provided by Eclipse itself) to help me complete my tasks.	1	2	3	4	5	6	7
I prefer using a suite of general purpose tools that I can apply to many tasks.	1	2	3	4	5	6	7
I prefer targeted tools that are designed to help me accomplish specific tasks.	1	2	3	4	5	6	7

Comments:

Reuse Questions:	strongly disagree	disagree	somewhat disagree	no opinion	somewhat agree	agree	strongly agree
I have reused source code (any scope).	1	2	3	4	5	6	7
I have reused whole classes.	1	2	3	4	5	6	7
I have reused whole features.	1	2	3	4	5	6	7
The reuse process as outlined on page one is similar to how I think of the reuse process.	1	2	3	4	5	6	7
When I reuse source code I am careful to only reuse exactly those portions that are relevant to the task I am completing.	1	2	3	4	5	6	7
When reusing code I worry about not fully understanding the code I am reusing.	1	2	3	4	5	6	7
I rely on IDE tools to help me complete reuse tasks.	1	2	3	4	5	6	7
I reuse source code to save myself time.	1	2	3	4	5	6	7
I reuse source code to increase the reliability of my code.	1	2	3	4	5	6	7
I reuse source code to increase the robustness of my code.	1	2	3	4	5	6	7
Given the choice of implementing a feature or reusing one from an existing system, I would choose to roll my own (aka. not reuse).	1	2	3	4	5	6	7
The size of the feature I am working on would influence my decision to try to reuse it or re-implement it from scratch.	1	2	3	4	5	6	7
Keeping track of the relevant details of a piece of source code while navigating its text can be difficult.	1	2	3	4	5	6	7
Understanding what dependencies a feature has on its context is important for me to determine whether I should reuse it.	1	2	3	4	5	6	7
I am more inclined to reuse smaller features about which I have a complete understanding, than larger features that are harder to reason about.	1	2	3	4	5	6	7
I prefer reusing smaller features because it is difficult to both build an understanding of a complex reuse task, and carry it out.	1	2	3	4	5	6	7
Reusing smaller features provides less benefit to me than reusing large features.	1	2	3	4	5	6	7
The definition of reuse outlined on page one is reasonable to me.	1	2	3	4	5	6	7

Comments:

9. Do you have any follow up questions / comments? Specifically, any comments about how you reuse code and the processes you use to investigate, evaluate, and carry out the reuse process are welcomed. Might you reuse code more often with the proper tool support? What would that support look like? Of course, any other comments would also be appreciated.

____

____

____

____

____

____

____

____

____

____

____

____

Thank you for your help with this study. I really appreciate your taking the time to fill it out and help me try to direct my research in such a way that it can hopefully be applicable to you and the work you do.

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Development												
D1	7	5	6	7	6	5	6	6	2	6	6	2
D2	2	3	3	7	7	5	3	7	6	7	5	6
D3	6	7	7	1	5	6	5	7	6	7	7	3
D4	1	6	7	7	5	6	5	7	6	4	6	3
D5	5	6	3	7	7	4	2	7	6	6	5	5
D6	7	5	7	7	7	5	4	7	6	5	5	3
D7	5	6	7	5	6	5	6	5	6	5	3	2
Environment												
E1	7	7	1	7	7	5	7	7	7	6	7	7
E2	7	7	1	7	2	3	6	3	6	5	7	5
E3	7	2	1	7	2	1	5	1	5	1	7	1
E4	7	2	1	7	1	1	5	1	4	1	7	1
E5	7	5	1	7	4	1	1	1	5	6	1	2
E6	6	6	4	5	5	5	4	5	3	6	5	6
E7	6	5	7	5	2	3	4	4	5	5	5	3
Reuse												
R1	6	7	7	7	7	7	6	7	7	7	7	7
R2	6	7	6	7	5	7	6	7	7	5	7	7
R3	4	7	3	3	3	7	2	7	7	5	3	2
R4	6	6	5	6	6	6	6	6	7	5	2	6
R5	5	3	6	6	7	7	6	6	5	5	6	6
R6	7	7	1	7	6	6	7	6	6	7	7	7
R7	6	6	1	7	2	1	2	2	5	5	7	1
R8	7	6	7	5	7	7	6	6	5	7	6	7
R9	7	4	7	6	6	6	6	6	6	7	6	7
R10	5	4	7	3	6	6	6	6	6	7	6	4
R11	2	3	1	1	5	2	4	5	3	2	3	2
R12	6	5	6	3	7	4	6	6	5	7	6	5
R13	6	6	2	5	6	4	6	6	7	7	6	5
R14	7	6	7	7	7	6	6	6	7	7	7	6
R15	5	3	7	5	7	6	7	5	6	7	6	6
R16	5	4	6	4	7	5	7	3	6	6	7	3
R17	6	5	1	1	4	4	1	3	4	3	6	2
R18	7	6	6	6	6	6	6	5	6	6	2	6

Table A.1: Pragmatic reuse industrial survey responses.

Question	Min	Max	Median	Mode	Avg	Stddev(Avg)
Development						
D1	2	7	6	6	5.33	1.60
D2	2	7	6	7	5.08	1.80
D3	1	7	6	7	5.58	1.80
D4	1	7	6	6	5.25	1.74
D5	2	7	6	5	5.25	1.53
D6	3	7	6	7	5.67	1.31
D7	2	7	5	5	5.08	1.32
Environment						
E1	1	7	7	7	6.25	1.69
E2	1	7	6	7	4.92	2.06
E3	1	7	2	1	3.33	2.53
E4	1	7	2	1	3.17	2.54
E5	1	7	3	1	3.42	2.40
E6	3	6	5	5	5.00	0.91
E7	2	7	5	5	4.50	1.32
Reuse						
R1	6	7	7	7	6.83	0.37
R2	5	7	7	7	6.42	0.76
R3	2	7	4	7	4.42	1.98
R4	2	7	6	6	5.58	1.19
R5	3	7	6	6	5.67	1.03
R6	1	7	7	7	6.17	1.62
R7	1	7	4	1	3.75	2.35
R8	5	7	7	7	6.33	0.75
R9	4	7	6	6	6.17	0.80
R10	3	7	6	6	5.50	1.19
R11	1	5	3	2	2.75	1.30
R12	3	7	6	6	5.50	1.12
R13	2	7	6	6	5.50	1.32
R14	6	7	7	7	6.58	0.49
R15	3	7	6	7	5.83	1.14
R16	3	7	6	6	5.25	1.42
R17	1	6	4	1	3.33	1.75
R18	2	7	6	6	5.67	1.18

Table A.2: Pragmatic reuse industrial survey aggregate responses.

Gilligan Tool Questionnaire

Please email responses to rtholmes@cs.ucalgary.ca

Thanks for trying out my tool. These questions are designed to be answered after filling out the pre-questionnaire and trying out my tool on some tasks that are of interest to you. Please fill out one reuse task page for each task you attempted! When you email me your responses, include this document as well as the XML stats files created by the tool (will be *-stats.xml in the directory you set as your persistence location in the wizard).

Name:

Overall Comments:

1. Did Gilligan improve your ability to plan your reuse tasks?

2. Was Gilligan able to give you a global overview of your reuse tasks?

3. Did Gilligan help you decide whether or not to proceed with the reuse tasks?

4. What aspects of Gilligan were most helpful for you?

5. What aspects of Gilligan were least helpful?

6. How can Gilligan be improved to help you most with your reuse tasks?

Reuse Task

1. What did this task consist of?

2. Have you attempted this task in the past?

3. How large was your reuse task (in # of units)?

	Methods	Classes	Packages
Accept			
Reject			
Remap			
Already Provided			

4. How many lines of code were reused in this task?

Task Questions:	strongly disagree	disagree	somewhat disagree	no opinion	somewh at agree	agree	strongly agree
Gilligan helped me understand this task	1	2	3	4	5	6	7
Gilligan helped me decide whether or not to proceed with the reuse activity	1	2	3	4	5	6	7
Gilligan helped me discover the dependencies in the code	1	2	3	4	5	6	7
I was able to make decisions (accept, reject, etc) based on the information provided by the tool	1	2	3	4	5	6	7
Using Gilligan could help me attempt larger, more complex reuse tasks	1	2	3	4	5	6	7

5. Do you have any other comments?

Appendix B

Sample Graph Layout Algorithms

This appendix contains four examples of graph layout strategies we investigated for the first iteration of the Gilligan prototype (Chapter 6). We decided that Figure B.4 would be best suited to our needs. The notations that have been added to the figures have been lost to the sands of time.

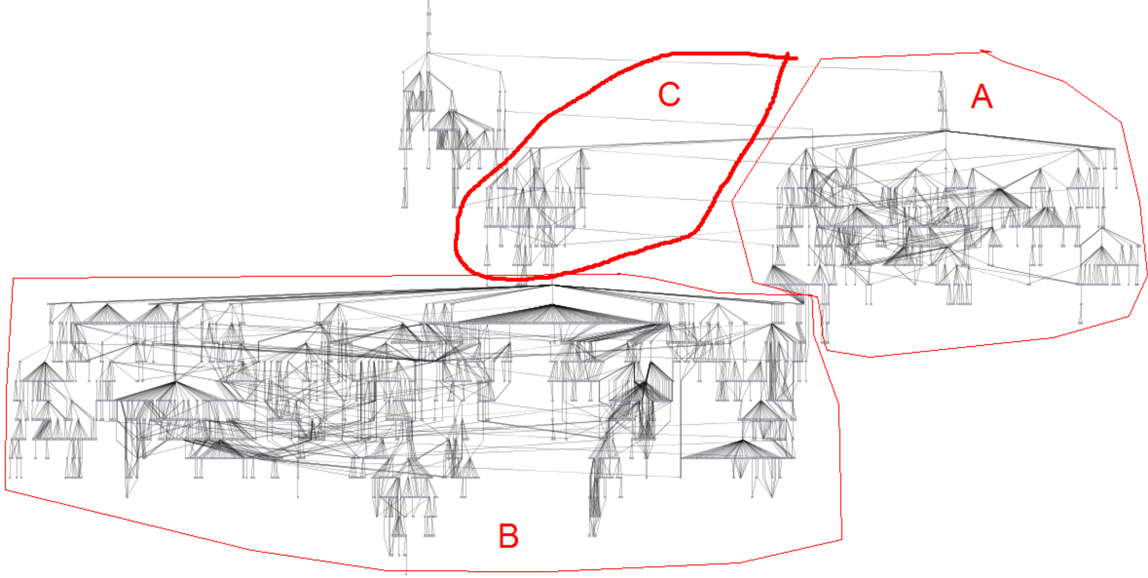


Figure B.1: Tree layout.

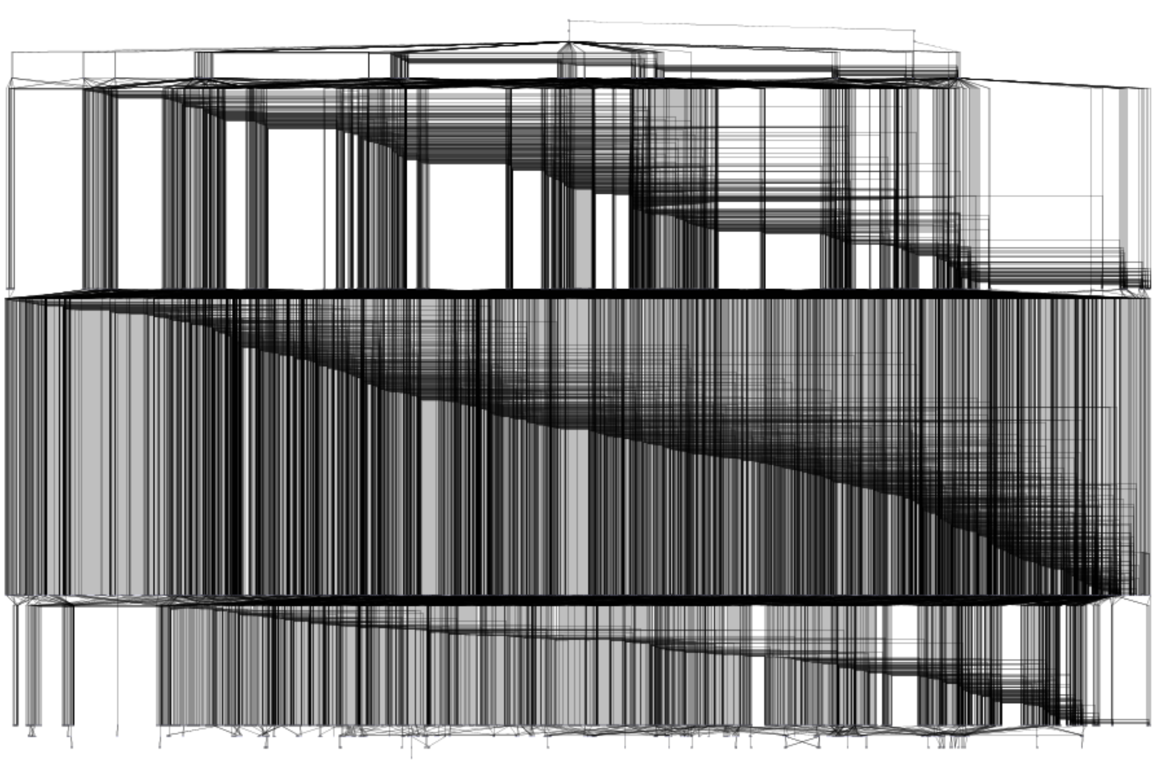


Figure B.2: Tree layout enforcing a depth hierarchy.

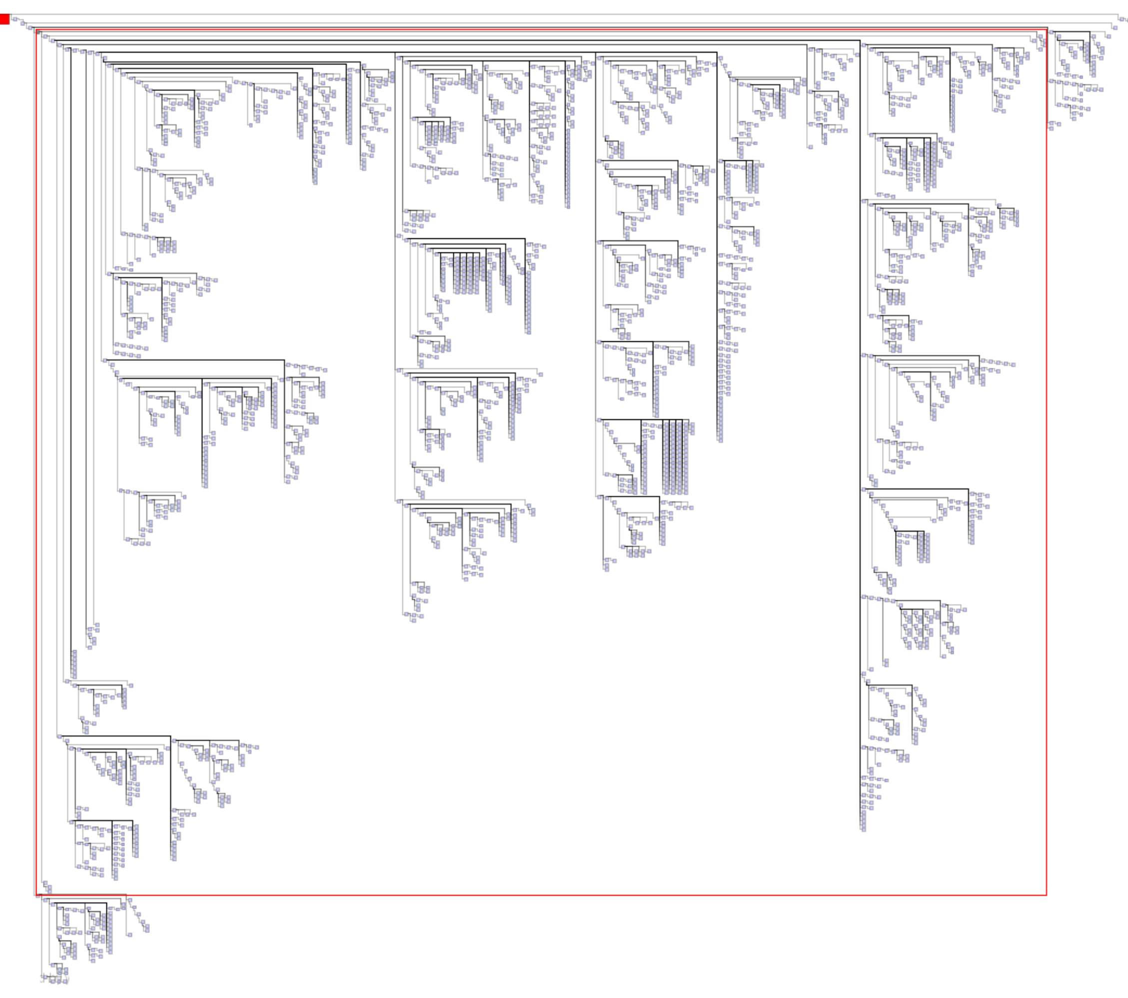


Figure B.3: Box-style tree layout.

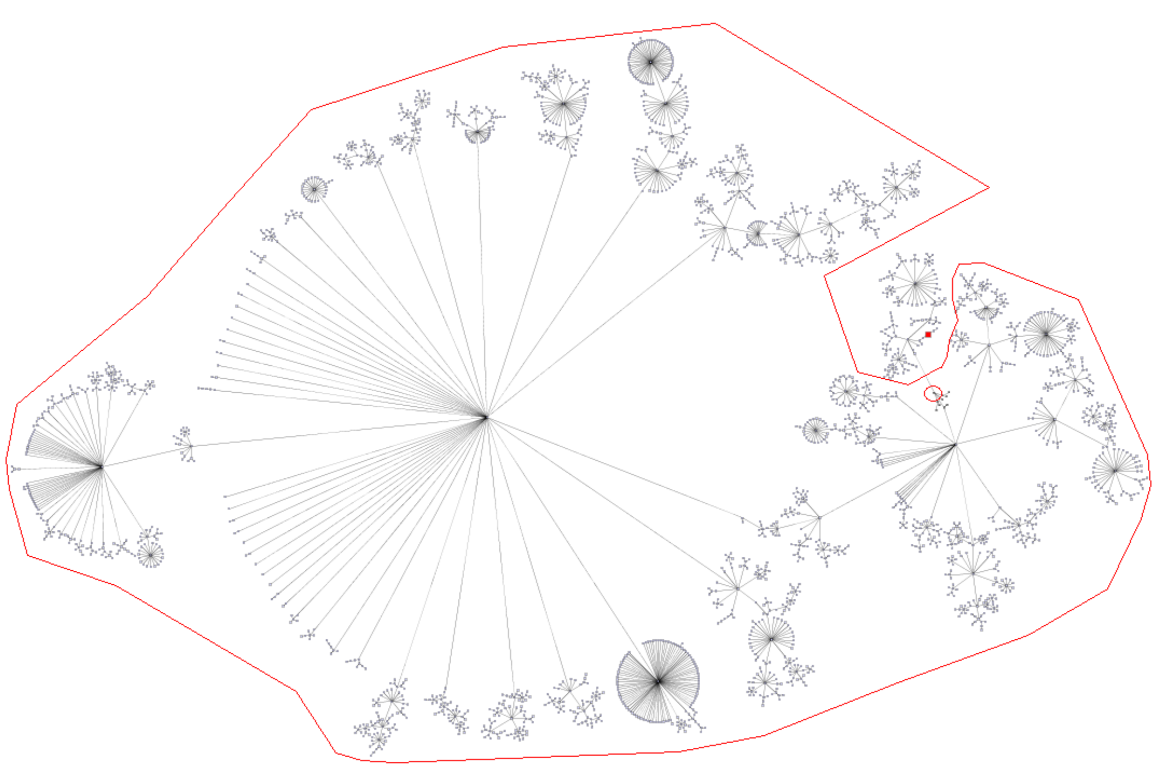


Figure B.4: Radial layout.

Appendix C

Planning evaluation

The laboratory materials for the case study described in Section 6.3 is provided in Section C.1 (p. 153).

Initial Gilligan Evaluation

Overview

Thank you for participating in the evaluation of Gilligan, the reuse tool I am developing. The ultimate goal of the Gilligan system is to help you navigate source code abstractly to enable you to formulate a reuse plan. This plan can then be used to automatically migrate the code you want to reuse into your system.

The current prototype of Gilligan focuses on the first part of that plan: it is designed to help you navigate through the source code in a system and decide which portions are relevant to you and which are not. The goal of this evaluation is to determine if the visual metaphor we are using for the code enables developers to have fine-grained enough control to make relevant decisions without being overwhelmed by the complexity of the software they are trying to reuse.

What do we want to find out?

The primary question we're trying to answer at this stage is:

- Can developers use Gilligan effectively to make reasonable reuse decisions/plans?

Specific sub-questions include:

- Using the visual tools provided, can I navigate the source code effectively to both gain the fine-grain understanding necessary while allowing me to abstract away unnecessary details?
- Can I decide whether or not to reuse this source code feature? Why or why not?
- Does encoding the decisions I have made about the dependencies within the system into the diagram make it easier to understand the reuse task I am attempting to perform?

What we want to do is make a case that using Gilligan, as it currently exists, developers can plan out reuse tasks better than doing it manually. We want to use this finding to setup the rest of our work that will support the extraction of the feature its integration into your target code. This work will be forthcoming.

What do we want you to do?

Ultimately, we want to know what your experience using Gilligan is like. Specifically, we'd like to know how the answers to the three questions above in relationship to how you used the tool. How was it successful? How did it fail? How can it be improved? The more specific, the better.

The intent of this evaluation is for you to use the tool on cases of reuse that you are investigating at work. If you're not currently looking at reusing any code, try it out on you code base by trying to see if you can use the tool identify and manage a feature within the code. Please try it on a few different tasks, of different scopes, we'd really like to know where this approach will work and where it won't.

When you're done you can send us some logs that the tool will make of your progress so we can measure the size of the problems you investigated and how the tool performed for them.

How to install Gilligan

We have created an Eclipse update site for Gilligan.

1. Open Eclipse and using the help menu: go to "Help->Software Updates->Find and Install".
2. Select "Search for New Features to Install" and press next.
3. Press the "New Remote Site" button. Call the update site "Gilligan" and set the site to be:

   http://pages.cpsc.ucalgary.ca/~rtholmes/gilligan/ And press "OK".
4. Select the Gilligan update site you just created and press finish; follow the remaining prompts.

* If you get a message about Gilligan requiring the Graphical Editor Framework (GEF), you can install install it by selecting the Eclipse.org update site (use the Callisto discovery site with 3.2) as well as Gilligan in the last step. You can then choose GEF under Graphical Editors, in addition to the Gilligan components.

How to use the tool

Using Gilligan comes down to a few primary steps:

1. Start a Gilligan Session
2. Navigate the graph, making decisions about the various nodes and edges as you go
3. Determine whether or not you would proceed with the reuse task
4. Save your session and send it back to me with your feedback

If you want to see what a Gilligan task looks like after a few minutes of navigation click here . In this example I am navigating a graph drawing feature from within Azureus that I want to reuse within a GPS application.

The Wizard

Gilligan is always started through a wizard that is activated by choosing the "Start Gilligan" button on the toolbar. In the wizard you will be asked to select the project that contains the code you want to reuse as well as the project you want to reuse the code within. This will allow Gilligan to extract the structural dependencies within the source system as well as identify those that are similar to your target environment.

On the "Select Element" page of the wizard where you are presented with a tree view of the packages, classes and methods in your source system it is important that you **select only one class** as your starting context. This is just a tooling limitation but if you select a package, method, or a class that isn't actually within your project (such as java.lang.Integer) the tool won't work correctly.

On the Select Element page you're really selecting the point from which you want to start navigating the feature you want to reuse, so choose a class that you think is important to the task you are trying to accomplish. If you're only interested in a method or a few methods, don't worry about it, you can start from the class and only investigate those methods you are interested in.

The Perspective

The Gilligan perspective will open up once you've selected your starting point. This perspective contains 5 parts. The leftmost navigator view will show a tree view of all of the nodes you have visited. Even if you make some go away on the picture they will still be there (and can be searched with the box at the top

(wildcards allowed)). Below that is a properties box that allows you to view specifics about a node, as well as edit some of its properties. The main view (top right) provides a graphical depiction of the source code you're looking at, while below that is the source view (source for the node or edge you've selected) and the reuse summary view.

Using the Graphical View

The main view shows a graphical depiction of the class you initially selected. Nodes in this abstraction can be either packages, classes, or methods. Edges between nodes can represent declares relationships (sold gray line with an unclosed arrow), calls relationships (dotted black lines with small sold arrow), or inheritance relationships (blue line with a large solid arrow).

Node Colours: The graphical view is supposed to give you a visual depiction of both the code and the decisions you made about different parts of the system. As you look at nodes you can classify them into one of three categories:

1. Accept: You want to reuse this code, it is applicable to your task.
2. Reject: This node is not at all useful to you, you don't want it reused.
3. Remap: This node implements functionality you need but that you want to do in a different way. You will remap this to existing code in your system (or you will write new code to meet this dependency).

A fourth state, Already Provided, also exists. This state means that the dependency from the source system is already met in the target (for instance anything marked java.*). This is more flexible than it looks. If you Accept node A which had dependencies on B and C but you only want B but not C, we can treat references to B from within A specially when it comes to extracting the code. You can select the state of a node either through the context menu, the properties view, or by selecting the appropriate box when hovering over a node.

Navigation: You can navigate through a nodes dependencies either by double clicking on them, or though the context menu. The primary navigation actions are open (shows a nodes dependencies) and collapse (collapse a nodes children into itself or collapse the node into its parent). When you hover over a node you can also click on the X to collapse, up arrow to see classes/interfaces that are extended/implemented, and the down arrow to see classes/interfaces that extend/implement the current node).

Edges: You can double click on most edges to get the source that realizes that edge. For instance, if you double click on the edge representing a call from method A to method B you will be presented with the source for method A highlighted by the calls to method B. This can be handy to show you _specifically_ why an edge exists between nodes so you don't need to read the whole method and figure it out yourself. This also works if you click on an edge from a class to a package for instance to see all the portions of that class that depend on the package. This doesn't work for all edges, we're working on this.

Tips: The most important thing using the visual view is to try to avoid becoming overwhelmed by all of the nodes and edges that are on the screen. The tool tips provided when you hover over a node gives you a good indication of how 'busy' that node is (number of classes / methods used). Usually, it is best to collapse yellow nodes down to their package level, unless those nodes are particularly interesting to you. This gets them out of the way as you don't really need to act on them. Similarly, if you choose to reject functionality it is usually helpful to collapse them down to the class or package level if the diagram is getting complicated.

PLEADING VOICE Please try not to get too distracted by the current layout situation. Currently there are some bugs causing nodes to overlap one and other. Also, parent nodes tend to gravitate toward the center of the screen which causes things to get a little muddled. It can often be helpful to drag nodes out of the way when they're there. These problems will be fixed, but we're more interested in providing actual extraction and integration features at this point than fixing these bugs.

Using the other views

The source code view shows you the source for any node (click on the S box when hovering). It also shows a highlighted version of the source if you click on an edge. The summary view gives a simple overview of the size of the reuse task. This is a last minute feature so at the moment it doesn't have a whole lot of functionality. Specific feedback about the types of information you'd like to see in this view is appreciated.

How to submit your experiences

1. Start a Gilligan Task
2. Explore the code, annotating it as you go
3. Save the Gilligan Session
4. Email me your -stats.xml file, as well as your impressions for that task. did it work? did it not? Feel free to send multiple -stats.xml files and your impressions at once if you'd prefer.

When you start a new Gilligan task it automatically saves a file to the persistence location you selected in the wizard. However, it is important that you press the 'Save Gilligan Session' button at the end of your work as well so it can record some statistics (how many nodes you looked at, how many you decorated etc). These statistics are saved in the same folder as the persistence file you chose (say "c:\task1.gil") but with "-stats.xml" appended to the end.

Limitations of the tool

The prototype version of Gilligan has a number of limitations. While these can be annoying, please try to see past them when you are using the tool, we are more interested in limitations of the approach rather than limitations of our tooling. However, we do want to improve the tool so please make any suggestions you'd like to see.

- Gilligan can only be opened once without restarting the IDE (this is horrible, I know)
- Nodes on the visual diagram occlude each other
- The visual layout is frustrating. Particularly, types tend to gravitate to the center and overlap
- Sometimes the animations can be a little overwhelming, particularly when many nodes are being added / collapsed
- No undo/redo
- LOC isn't computed work in the reuse summary
- Feature extraction / integration not implemented

Appendix D

Visualization evaluation

The data table used to construct the graphs presented in Section 7.3 is given in Table D.5. The 'gold standard' result set we used to assess precision and recall is also included for each task (Table D.1, Table D.2, Table D.3, Table D.4).

net.sf.ganymede.Ganymede
net.sf.ganymede.Ganymede.getDefault()
net.sf.ganymede.Ganymede.plugin
net.sf.ganymede.GanymedeUtilities
net.sf.ganymede.GanymedeUtilities.getSite()
net.sf.ganymede.GanymedeUtilities.getStartAction()
net.sf.ganymede.GanymedeUtilities.getStopAction()
net.sf.ganymede.GanymedeUtilities.initActions()
net.sf.ganymede.GanymedeUtilities.isActionsInited()
net.sf.ganymede.GanymedeUtilities.mActionsInited
net.sf.ganymede.GanymedeUtilities.mSite
net.sf.ganymede.GanymedeUtilities.mStartAction
net.sf.ganymede.GanymedeUtilities.mStopAction
net.sf.ganymede.GanymedeUtilities.setActionsInited(Z)
net.sf.ganymede.GanymedeUtilities.setStartAction(Lorg.eclipse.jface.action.IAction;)
net.sf.ganymede.GanymedeUtilities.setStopAction(Lorg.eclipse.jface.action.IAction;)
net.sf.ganymede.log4j.Log4jServer
net.sf.ganymede.log4j.Log4jServer.<init>()
net.sf.ganymede.log4j.Log4jServer.getLog4jServer()
net.sf.ganymede.log4j.Log4jServer.getServerSocket()
net.sf.ganymede.log4j.Log4jServer.isServerUp()
net.sf.ganymede.log4j.Log4jServer.mLog4jServer

net.sf.ganymede.log4j.Log4jServer.mServerSocket
net.sf.ganymede.log4j.Log4jServer.mServerUp
net.sf.ganymede.log4j.Log4jServer.setLog4jServer(Lnet.sf.ganymede.log4j.Log4jServer;)
net.sf.ganymede.log4j.Log4jServer.setServerSocket(Ljava.net.ServerSocket;)
net.sf.ganymede.log4j.Log4jServer.setServerUp(Z)
net.sf.ganymede.log4j.Log4jServer.startListener()
net.sf.ganymede.preferences.Log4jPreferencePage
net.sf.ganymede.preferences.Log4jPreferencePage.P_PORT

Table D.1: Answer key for assessing precision and recall for Task 1. These structural elements represent the transitive closure of the structural elements of `Log4JServer.startListener()`, excluding the common dependencies between the source and target systems.

org.apache.commons.httpclient.Cookie
org.apache.commons.httpclient.Cookie.<init>(...)
org.apache.commons.httpclient.Cookie.LOG
org.apache.commons.httpclient.Cookie.cookieComment
org.apache.commons.httpclient.Cookie.cookieDomain
org.apache.commons.httpclient.Cookie.cookieExpiryDate
org.apache.commons.httpclient.Cookie.cookiePath
org.apache.commons.httpclient.Cookie.hasDomainAttribute
org.apache.commons.httpclient.Cookie.hasPathAttribute
org.apache.commons.httpclient.Cookie.isSecure
org.apache.commons.httpclient.Cookie.setComment(Ljava.lang.String;)
org.apache.commons.httpclient.Cookie.setDomain(Ljava.lang.String;)
org.apache.commons.httpclient.Cookie.setDomainAttributeSpecified(Z)
org.apache.commons.httpclient.Cookie.setExpiryDate(Ljava.util.Date;)
org.apache.commons.httpclient.Cookie.setPath(Ljava.lang.String;)
org.apache.commons.httpclient.Cookie.setPathAttributeSpecified(Z)
org.apache.commons.httpclient.Cookie.setSecure(Z)
org.apache.commons.httpclient.HeaderElement
org.apache.commons.httpclient.HeaderElement.<init>()
org.apache.commons.httpclient.HeaderElement.<init>(...)
org.apache.commons.httpclient.HeaderElement.<init>([C)
org.apache.commons.httpclient.HeaderElement.<init>([C,I,I)
org.apache.commons.httpclient.HeaderElement.LOG
org.apache.commons.httpclient.HeaderElement.getParameters()
org.apache.commons.httpclient.HeaderElement.parameters

org.apache.commons.httpclient.HeaderElement.parseElements([C)
org.apache.commons.httpclient.NameValuePair
org.apache.commons.httpclient.NameValuePair.<init>(...)
org.apache.commons.httpclient.NameValuePair.getName()
org.apache.commons.httpclient.NameValuePair.getValue()
org.apache.commons.httpclient.NameValuePair.name
org.apache.commons.httpclient.NameValuePair.setName(Ljava.lang.String;)
org.apache.commons.httpclient.NameValuePair.setValue(Ljava.lang.String;)
org.apache.commons.httpclient.NameValuePair.toString()
org.apache.commons.httpclient.NameValuePair.value
org.apache.commons.httpclient.cookie.CookieSpec
org.apache.commons.httpclient.cookie.CookieSpec.PATH_DELIM
org.apache.commons.httpclient.cookie.CookieSpecBase
org.apache.commons.httpclient.cookie.CookieSpecBase.LOG
org.apache.commons.httpclient.cookie.CookieSpecBase.datepatterns
org.apache.commons.httpclient.cookie.CookieSpecBase.parse(...)
org.apache.commons.httpclient.cookie.CookieSpecBase.parseAttribute(...)
org.apache.commons.httpclient.util.DateUtil
org.apache.commons.httpclient.util.DateUtil.DEFAULT_PATTERNS
org.apache.commons.httpclient.util.DateUtil.DEFAULT_TWO_DIGIT_YEAR_START
org.apache.commons.httpclient.util.DateUtil.parseDate(...)
org.apache.commons.httpclient.util.DateUtil.parseDate(...)
org.apache.commons.httpclient.util.ParameterParser
org.apache.commons.httpclient.util.ParameterParser.<init>()
org.apache.commons.httpclient.util.ParameterParser.chars
org.apache.commons.httpclient.util.ParameterParser.getToken(Z)
org.apache.commons.httpclient.util.ParameterParser.hasChar()
org.apache.commons.httpclient.util.ParameterParser.i1
org.apache.commons.httpclient.util.ParameterParser.i2
org.apache.commons.httpclient.util.ParameterParser.isOneOf(C,[C)
org.apache.commons.httpclient.util.ParameterParser.len
org.apache.commons.httpclient.util.ParameterParser.parse([C,I,I,C)
org.apache.commons.httpclient.util.ParameterParser.parseQuotedToken([C)
org.apache.commons.httpclient.util.ParameterParser.parseToken([C)
org.apache.commons.httpclient.util.ParameterParser.pos

Table D.2: Answer key for assessing precision and recall for Task 2. These structural elements represent the transitive closure of the structural elements of `CookieSpecBase.parse(..)`, excluding the common dependencies between the source and target systems.

net.sourceforge.ganttproject.GanttCalendar
net.sourceforge.ganttproject.GanttCalendar.<init>(I,I,I)
net.sourceforge.ganttproject.GanttCalendar.Clone()
net.sourceforge.ganttproject.GanttCalendar.add(I)
net.sourceforge.ganttproject.GanttCalendar.getDay()
net.sourceforge.ganttproject.GanttCalendar.getMonth()
net.sourceforge.ganttproject.GanttCalendar.getYear()
net.sourceforge.ganttproject.GanttCalendar.isFixed
net.sourceforge.ganttproject.GanttCalendar.isFixed()
net.sourceforge.ganttproject.GanttCalendar.setFixed(Z)
net.sourceforge.ganttproject.GanttProject
net.sourceforge.ganttproject.GanttProject$1TaskManagerConfigImpl
net.sourceforge.ganttproject.GanttProject$1TaskManagerConfigImpl.getCalendar()
net.sourceforge.ganttproject.GanttProject.HUMAN_RESOURCE_MANAGER_ID
net.sourceforge.ganttproject.GanttProject.ROLE_MANAGER_ID
net.sourceforge.ganttproject.GanttProject.getActiveCalendar()
net.sourceforge.ganttproject.GanttProject.getHumanResourceManager()
net.sourceforge.ganttproject.GanttProject.getRoleManager()
net.sourceforge.ganttproject.GanttProject.managerHash
net.sourceforge.ganttproject.GanttProject.myFakeCalendar
net.sourceforge.ganttproject.GanttProject.myUIConfiguration
net.sourceforge.ganttproject.calendar.GPCalendar
net.sourceforge.ganttproject.calendar.GPCalendar.isNonWorkingDay(...)
net.sourceforge.ganttproject.task.Task
net.sourceforge.ganttproject.task.Task.getManager()
net.sourceforge.ganttproject.task.TaskImpl
net.sourceforge.ganttproject.task.TaskImpl.getManager()
net.sourceforge.ganttproject.task.TaskImpl.myManager
net.sourceforge.ganttproject.task.TaskManager
net.sourceforge.ganttproject.task.TaskManager.getCalendar()
net.sourceforge.ganttproject.task.TaskManagerConfig
net.sourceforge.ganttproject.task.TaskManagerConfig.getCalendar()
net.sourceforge.ganttproject.task.TaskManagerImpl

net.sourceforge.ganttproject.task.TaskManagerImpl.getCalendar()
net.sourceforge.ganttproject.task.TaskManagerImpl.getConfig()
net.sourceforge.ganttproject.task.TaskManagerImpl.myConfig
net.sourceforge.ganttproject.task.dependency.TaskDependency
net.sourceforge.ganttproject.task.dependency.TaskDependency.getDependant()
net.sourceforge.ganttproject.task.dependency.TaskDependency.getDifference()
net.sourceforge.ganttproject.task.dependency.TaskDependencyImpl
net.sourceforge.ganttproject.task.dependency.TaskDependencyImpl.getDependant()
net.sourceforge.ganttproject.task.dependency.TaskDependencyImpl.getDifference()
net.sourceforge.ganttproject.task.dependency.TaskDependencyImpl.myDependant
net.sourceforge.ganttproject.task.dependency.TaskDependencyImpl.myDifference
net.sourceforge.ganttproject.task.dependency.constraint.ConstraintImpl
net.sourceforge.ganttproject.task.dependency.constraint.ConstraintImpl.addDelay(...)
net.sourceforge.ganttproject.task.dependency.constraint.ConstraintImpl.myDependency
net.sourceforge.ganttproject.time.gregorian.GregorianCalendar
net.sourceforge.ganttproject.time.gregorian.GregorianCalendar.<init>(I,I,I)
net.sourceforge.ganttproject.time.gregorian.GregorianCalendar.add(I,I)

Table D.3: Answer key for assessing precision and recall for Task 3. These structural elements represent the transitive closure of the structural elements of `ConstraintImpl.addDelay(..)`, excluding the common dependencies between the source and target systems.

org.apache.commons.collections.bidimap.TreeBidiMap
org.apache.commons.collections.bidimap.TreeBidiMap.<init>()
org.apache.commons.collections.bidimap.TreeBidiMap.get(Ljava.lang.Object;)
org.jajuk.Main
org.jajuk.Main.bIdeMode
org.jajuk.base.FileManager
org.jajuk.base.FileManager$1
org.jajuk.base.FileManager$1.<init>()
org.jajuk.base.FileManager$2
org.jajuk.base.FileManager$2.<init>()
org.jajuk.base.FileManager.<init>()
org.jajuk.base.FileManager.alBestofFiles
org.jajuk.base.FileManager.alNovelties
org.jajuk.base.FileManager.bRateHasChanged
org.jajuk.base.FileManager.getInstance()
org.jajuk.base.FileManager.rateComparator

org.jajuk.base.FileManager.scoreComparator
org.jajuk.base.FileManager.singleton
org.jajuk.base.History
org.jajuk.base.History.addItem(Ljava.lang.String;,J)
org.jajuk.base.History.vHistory
org.jajuk.base.HistoryItem
org.jajuk.base.HistoryItem.<init>(Ljava.lang.String;,J)
org.jajuk.base.HistoryItem.getFileId()
org.jajuk.base.HistoryItem.lDate
org.jajuk.base.HistoryItem.sFileId
org.jajuk.base.IPropertyable
org.jajuk.base.ItemManager
org.jajuk.base.ItemManager.<init>()
org.jajuk.base.ItemManager.bLock
org.jajuk.base.ItemManager.getItem(Ljava.lang.String;)
org.jajuk.base.ItemManager.getLock()
org.jajuk.base.ItemManager.hmItems
org.jajuk.base.ItemManager.hmPropertiesMetaInformation
org.jajuk.base.ItemManager.registerProperty(...)
org.jajuk.base.PropertyMetaInformation
org.jajuk.base.PropertyMetaInformation.<init>(...)
org.jajuk.base.PropertyMetaInformation.bConstructor
org.jajuk.base.PropertyMetaInformation.bCustom
org.jajuk.base.PropertyMetaInformation.bEditable
org.jajuk.base.PropertyMetaInformation.bMergeable
org.jajuk.base.PropertyMetaInformation.bShouldBeDisplayed
org.jajuk.base.PropertyMetaInformation.cType
org.jajuk.base.PropertyMetaInformation.format
org.jajuk.base.PropertyMetaInformation.getName()
org.jajuk.base.PropertyMetaInformation.oDefaultValue
org.jajuk.base.PropertyMetaInformation.sHumanType
org.jajuk.base.PropertyMetaInformation.sName
org.jajuk.i18n.Messages
org.jajuk.i18n.Messages$1
org.jajuk.i18n.Messages$1.<init>()
org.jajuk.i18n.Messages.<init>()
org.jajuk.i18n.Messages.alDescs

org.jajuk.i18n.Messages.alLocals
org.jajuk.i18n.Messages.getErrorMessage(Ljava.lang.String;)
org.jajuk.i18n.Messages.getInstance()
org.jajuk.i18n.Messages.getProperties()
org.jajuk.i18n.Messages.getPropertiesEn()
org.jajuk.i18n.Messages.getString(Ljava.lang.String;)
org.jajuk.i18n.Messages.isInitialized()
org.jajuk.i18n.Messages.mesg
org.jajuk.i18n.Messages.parseLangpack(Ljava.lang.String;)
org.jajuk.i18n.Messages.properties
org.jajuk.i18n.Messages.propertiesEn
org.jajuk.i18n.Messages.sLocal
org.jajuk.util.ConfigurationManager
org.jajuk.util.ConfigurationManager.getProperty(Ljava.lang.String;)
org.jajuk.util.ConfigurationManager.properties
org.jajuk.util.ITechnicalStrings
org.jajuk.util.ITechnicalStrings.CONF_HISTORY
org.jajuk.util.ITechnicalStrings.DATE_FORMAT_2
org.jajuk.util.ITechnicalStrings.FEEDBACK_LINES
org.jajuk.util.ITechnicalStrings.FILE_LANGPACK_PART1
org.jajuk.util.ITechnicalStrings.FILE_LANGPACK_PART2
org.jajuk.util.ITechnicalStrings.MAX_HISTORY_SIZE
org.jajuk.util.ITechnicalStrings.XML_DIRECTORY
org.jajuk.util.ITechnicalStrings.XML_FILE_DATE
org.jajuk.util.ITechnicalStrings.XML_ID
org.jajuk.util.ITechnicalStrings.XML_NAME
org.jajuk.util.ITechnicalStrings.XML_QUALITY
org.jajuk.util.ITechnicalStrings.XML_SIZE
org.jajuk.util.ITechnicalStrings.XML_TRACK
org.jajuk.util.Util
org.jajuk.util.Util.getExecLocation()
org.jajuk.util.Util.sExecLocation
org.jajuk.util.log.Log
org.jajuk.util.log.Log.alSpool
org.jajuk.util.log.Log.debug(Ljava.lang.String;)
org.jajuk.util.log.Log.error(Ljava.lang.String;,Ljava.lang.String;,Ljava.lang.Throwable;)
org.jajuk.util.log.Log.error(Ljava.lang.Throwable;)

org.jajuk.util.log.Log.logger
org.jajuk.util.log.Log.spool(Ljava.lang.String;)
org.jajuk.util.log.Log.spool(Ljava.lang.Throwable;)

Table D.4: Answer key for assessing precision and recall for Task 4. These structural elements represent the transitive closure of the structural elements of `History.addItem(..)`, excluding the common dependencies between the source and target systems.

	Task 1		Task 2		Task 3		Task 4	
	Manual	Giligan	Manual	Giligan	Manual	Giligan	Manual	Giligan
Time								
	19	7	23	3	12	10	4	10
	11	3	17	3	11	9	10	19
	11	6	12	25	28	6	15	5
Average	13.67	5.33	17.33	10.33	17	8.33	9.67	11.33
Recall								
	0.7	1	0.42	1	0.28	0.5	0.15	1
	0.87	1	0.42	1	0.38	0.5	0.29	0.94
	0.73	1	0.53	0.98	0.52	0.5	0.14	1
Average	0.77	1	0.46	0.99	0.39	0.5	0.19	0.98
Precision								
	0.53	1	0.93	0.87	1	1	1	1
	0.73	1	1	0.86	0.76	1	0.75	1
	1	1	1	0.98	1	1	0.93	1
Average	0.75	1	0.98	0.9	0.92	1	0.89	1
Aggregate Values								
	Manual	Giligan	Manual	Giligan	Manual	Giligan	Manual	Giligan
Average Recall	0.77	1	0.46	0.99	0.39	0.5	0.19	0.98
Average Time	13.67	5.33	17.33	10.33	17	8.33	9.67	11.33

Table D.5: Data values for time, recall, and precision for the second prototype evaluation (Section 7.3).

Appendix E

Enactment evaluation

Section E.1 specifies the nature of all of the edits that contributed to the results presented in Section 8.3.2. The pragmatic reuse plans provided to the participants in the experiment described in Section 8.3.3 are included in Figure E.3 and Figure E.4. The experimental instructions provided to our participants for this experimentare included in Section E.2 (p. 172).

E.1 Effort

E.1.1 Extracting the Metrics Lines-of-Code Calculator

Creating this pragmatic reuse plan involved balancing the flexibility provided by the Metrics plug-in's framework to support an array of measurements about source code against our desire to reuse only one of them. More than any other element, we wanted to be able to reuse the method `AbstractLinesOfCode.calculateNumberOfLines(String)`. This one method contains a token parser to count the number of lines of source code, taking into consideration white-space and comments. When we created the reuse plan we decided to include a total of eight classes. Reusing these classes balanced the requirements of `AbstractLinesOfCode` against our desire to keep the reuse task as compact as possible.

To better provide us with the needed data, we also included portions of `AbstractMetricSource` as this class provides handles for passing `IJavaElement` objects (Eclipse's key representation of Java source code entities) to the calculator. At the same time, `AbstractMetricSource` provides additional functionality for aggregating and storing metric scores, neither of which were useful to us and so, we wanted to discard them. Ultimately, this pragmatic reuse task involved reusing portions of 9 classes for a total of 265 loc. 498 loc were rejected from the reused classes in order to eliminate functionality we did not need.

To manually perform this task, the 8 classes were copied (8 copy operations), 6 of these (`Metric`, `Calculator`, `AbstractMetricSource`, and `AbstractLinesOfCode`, `TypeMetrics`, and `Max`) re-

quired manual editing in order to fully compile. 62 compiler errors result from copying the source classes into the target project.

Fixing `Metric` involved removing the reference to its supertype `Constants` which was not reused (1 edit). Also, its reference to `FRACTION_DIGITS` had to be remapped from its old target (inside `Constants`) to its new location (inside `AbstractLinesOfCode`) (1 edit). Finally, the import statements needed to be updated (1 edit). Resolving the problems in `Calculator` involved removing the reference to its supertype `Constants` (1 edit), as well as removing the `calculatorDescriptor` field and the two methods (`setDescriptor(Descriptor)` and `getDescriptor()`) that used this field (3 edits). We rejected this field and its two supporting methods because they did not provide any functionality we needed to reuse. The `getPreferences()` method was also removed for the same reason (1 edit). Again, the import statements were updated to reflect the changes to the code (1 edit). The most changes were made to `AbstractMetricSource` because, while this class provided functionality we wanted to reuse, it also provided many features we were not interested in reusing. First, we removed the supertype reference to `Constants` (1 edit). Then we removed 5 fields and 23 methods that were associated with functionality we were not reusing (28 edits). Next, we added a new field corresponding to the logging functionality we were using in our system (1 edit) and remapped the calls from Metric's original logger to our own (1 edit). Next we managed calls to methods that had been rejected; this required removing two blocks corresponding to method calls to `metricsInterrupts()` and `isWarningsEnabled()`, and one block corresponding to a reference to a rejected field `childHandles` (3 edits). Finally the import statements were updated (1 edit). The last class that needed modification was `AbstractLinesOfCode`. First, we extracted the field `FRACTION_DIGITS` from `Constants` (1 edit). This field was extracted so that it could be reused without having to reuse the remainder of `Constants` (which included 36 additional constants). Next, a field for our logging functionality was added (1 edit) and references to `logError` and `logMessage` were remapped to reference our added field instead of the default metrics logging harness (2 edits). Lastly, the import statements were updated (1 edit). In addition to these edits, one method `TypeMetrics.getCalculators()` needed to be modified to make the reused code function outside of the metrics system. This involved inserting a short 3 line block of code (1 edit).

In enacting this pragmatic reuse plan semi-automatically, Gilligan resolves all but two of the compilation errors. By trying to reject two overlapping statements, Gilligan created a code fragment that did not comply to the Java Language Specification. In this case both `isWarningsEnabled()` and `checkRange(Metric)` were rejected. We resolved the error by fixing the overlapping comments (1 edit). The code for the snippet is in Figure E.1, prior to the manual edit. Gilligan took less than 3 seconds to extract and integrate the code; the required manual edit took less than 1 minute to resolve.

The manual treatment of this task required 58 manual edits. Of these, only 7 were not purely mechanical. That is, for the 3 references to fields or methods that had been rejected, we actually had to make changes to bodies of methods. In the 4th instance, we had to remap the reference

```
/** GILLIGAN: Invocation rejected in reuse plan
if (MetricsPlugin.isWarningsEnabled()) {
     /** GILLIGAN: Invocation rejected in reuse plan
     checkRange(value);
     */
}
*/
```

Figure E.1: Error caused when Gilligan rejects structural elements that are nested in the source code.

to the extracted field `FRACTION_DIGITS`. Finally, 3 sets of calls to the logging functionality were remapped. The other 50 changes simply required us to copy code from the old system to the new one and remove fields and references we were not interested in reusing. Performing the manual task took approximately 20 minutes. In contrast, using Gilligan we only had to make 2 edits spending less than 2 minutes; the tool resolved 48 of 49 compiler errors.

E.1.2 Extracting the Azureus Network Throughput View

In the manual case, we had to copy the classes (6 copy operations) to the target environment; 97 compiler errors result from copying the source classes into the target project.

Of these, `BackgroundGraphic`, `ScaledGraphic`, and `SpeedGraphic` each needed modification as together they caused 26 compiler errors. `BackgroundGraphic` had a field of type `AEMonitor` that managed network locks in the core of Azureus. This functionality was not relevant to our task and so, it was rejected. After removing the field (1 edit), we also extracted two fields (`Colors.black` and `Colors.white`) into `SpeedGraphic` (2 edits). The references to these two fields were remapped using search and replace (`Colors.black` to `SpeedGraphic.black` and `Colors.white` to `SpeedGraphic.white`) (1 edit). Finally, the import statements were updated to replace the reference to `AEMonitor` with `SpeedGraphic` (1 edit). As we had already extracted `Colors.black`, we only needed to remap its reference (1 edit) and repair the import statements (1 edit) in `ScaledGraphic`. In `SpeedGraphic`, we first removed the supertype reference to `ParameterListener` (1 edit). We also removed the methods `getInstance()` and `parameterChanged()` (2 edits). As `SpeedGraphic` makes heavy use of fields from `Colors`, 9 more fields were extracted (9 edits) to `SpeedGraphic`, and their references remapped using search and replace (`Colors.xyz` with `SpeedGraphic.xyz`) (1 edit). As the `AEMonitor`-typed field had been removed from the supertype `BackgroundGraphic`, we also had to remove the 4 locations in the code that called methods on this field (4 edits). One call to `ParameterListener.parameterChanged()` was removed from the class constructor as well (1 edit). Finally, the import statements were updated (1 edit).

Gilligan performed the extraction and integration process in 3 seconds. 11 compiler errors resulted which required four edits to fix. One edit had to be applied to `SpeedGraphic` class to repair

an error caused by Gilligan when the fields were extracted. This problem is shown in Figure E.2; the field declaration order had been inverted so that BLUES_DARKEST was referenced before it was defined (1 edit). The second problem was that SpeedGraphic instantiates several custom colour fields in a static initializer. Currently, Gilligan cannot remap these references and as such we had to use search and replace to fix them manually (1 edit). These 2 errors took one minute to resolve.

```
/** GILLIGAN: Field extracted in reuse plan. */
static Color[] blues = new Color[BLUES_DARKEST + 1];

/** GILLIGAN: Field extracted in reuse plan. */
static final int BLUES_DARKEST = 9;
```

Figure E.2: Error caused by Gilligan not considering field ordering.

In total, 32 edits and 15 minutes were required to complete the task manually. These changes were less amenable to simple edit support than in the Metrics case study. This is because the changes involved more code modifications than removal. In the end, 10 fields were extracted and their references remapped. In addition, removing references to AEMonitor and ParameterListener required changing several references, declarations, and calls within BackgroundGraphic and SpeedGraphic. In contrast, Gilligan detected and remedied most of these situations.

Figure E.3: Pragmatic reuse plan for the Metrics task. Common dependencies have been elided to save space.

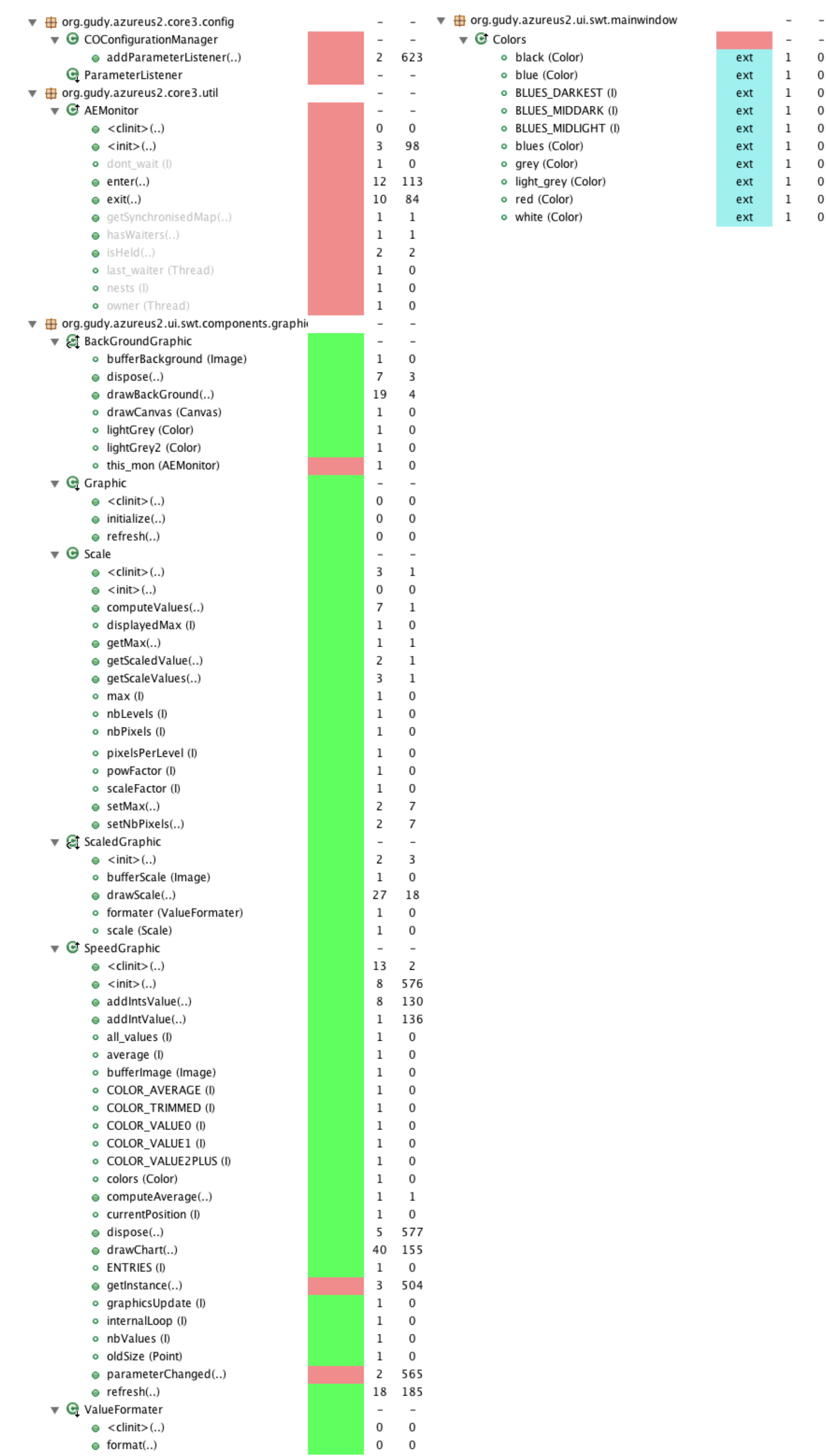


Figure E.4: Pragmatic reuse plan for the Azureus task. Common dependencies have been elided to save space.

Procrustes Summer 2007 Study Appendix

This page provides additional details of our Procrustes Summer 2007 pragmatic-reuse user experiment. If you have any questions, feel free to contact rtholmes@cs.ucalgary.ca. The content of this page is a slightly abbreviated version of the description given to the participants of the experiment. It has been modified to remove the download instructions for the tool itself.

To look at the reuse plans themselves you can look in this zip file

Gilligan/Procrustes Study: Quick Start Guide

The document is split into lots of sections so you can read it choose-your-own-adventure style. From now until you're done should take between one and two hours.

Table of Contents

1. Study overview
2. Preparing your environment
3. The tasks
    - Task 1
    - Task 2
4. The treatments
    - Automated enactment
    - Manual enactment
5. The tools
    - Gilligan reuse planning environment
    - Procrustes reuse plan enactment tool
6. Questionnaire

Study overview

This is a remote study, you will install the tool on your machine, perform 2 tasks, and send me the results. You can contact me at any time for information or to ask any questions. In this study you will be asked to undertake two tasks. If possible, try out the tools before you start your first task so you feel comfortable with how they work. After completing the tasks, please fill out the questionnaire given at the end of this page; this questionnaire is the most important part of the evaluation for me, please take your time and be as verbose as possible. Timing is important for this evaluation, so if possible, please pick two time-blocks where you could devote 1 hour each time. Just so you don't worry, 1 hour is an upper bound; some tasks and treatments are much faster.

When you're done you'll be sending me (rtholmes@cs.ucalgary.ca) **three** things (see the Submitting results section):

- Your questionnaire answers
- A zip of the Gilligan log files
- A zip of the two target workspaces from your tasks

If you have time, and feel so inclined, I'd also appreciate it if you'd try Gilligan/Procrustes on one of your own tasks. If you try it out on something of your own just send me a quick description of what it was and include your thoughts about how the tools helped/hindered you for this task. I welcome any of your feedback and suggestions.

Preparing your environment

My primary development environment is Eclipse 3.3 on the Mac. I've tried it on 3.3 on Windows and it should be OK and there's nothing in there that should stop it from working on Eclipse 3.2. If you're using 3.2, don't bother upgrading, it should be fine. If you suspect you have found any bugs, please let me know.

1. Download the study workspace projects.
2. Import the projects into Eclipse. For Eclipse 3.3 you can just go File->import->general->existing projects, choose 'select archive' and give it the study-workspace.zip file. Select all four projects and press import. This may work for 3.2 as well but I haven't tried it. If you prefer you can manually import each project into your workspace as well.
3. Install Gilligan/Procrustes using its update site. Help->Software Updates->Find and Install. Choose 'Search for new features to install' and select 'New Remote Site'. You can name the remote site anything you choose, in the url enter ***UPDATE SITE INFORMATION REMOVED***. Follow the install wizard through; this will download the necessary files and install them. A restart of Eclipse will be required to get the plug-in installed.
4. After restarting, start the Gilligan/Procrustes interface. Window->open perspective->other, choose 'Gilligan/Procrustes Reuse Environment'.

You now should be good to go. If you're going to be doing the eval on a plane or on the road with a laptop, save this page and you can head offline (until you send me the results).

The Tasks

The goal of Gilligan/Procrustes is to help you complete reuse tasks easier. As such, you will be performing two reuse tasks for this evaluation. Each task involves reusing source code (that was not designed or modularized for reuse) from an open-source project within your own (simulated by the target projects) project. To complete the tasks you will reuse the source code as described in the provided reuse plans and resolve any compilation errors (don't worry about warnings) in the target project and run a test harness against it to make sure the reused code works. You will be attempting each task using one of two treatments (as I specified in my email to you).

Task 1: Reusing the LOC calculator from metrics.sf.net plug-in

The goal of this task is to reuse the lines-of-code (LOC) calculator from the Eclipse metrics plug-in from metrics.sf.net. The metrics plug-in computes the LOC using the Eclipse AST and its associated source text. The plan for this task has been included in the task "LOC Calculator - net.sourceforge.metrics in net.sourceforge.metrics_target". Your source project for this task is the project net.sourceforge.metrics; the project you want to reuse the code in is net.sourceforge.metrics_target.

The target project contains a couple of utility classes: TheLog for logging functionality, LOCTest is a harness for evaluating the feature, and JavaModelUtilities, a utility class used by LOCTest. This harness is actually set up as an Eclipse plug-in itself. The easiest way to run it is to open the Manifest.mf file from the META-INF folder in the root directory of the project. Then click on the overview page tab (bottom left of the editor). On this page there should be a 'Testing' sub-section and you can click 'Launch an eclipse application in debug-mode'. This will start up a runtime workspace in which the harness will be running. If you don't have a runtime workspace available just select a new empty directory and it'll create one for you. Once the runtime workbench loads, you'll need to make sure you have a java project in there, either create a dummy project with some code (make sure it compiles and is not in the default package), or just import one of the projects (the metrics project itself is nice and is not too large) as described in the initial setup. The test harness should have added a button 'Test LOC Counter' to the menu bar. When pressed, this should go through all of the java source files in all of the projects in your runtime workspace and print out a LOC count for each one. This count will appear in the console view of your original workbench (aka. _not_ the runtime workbench). Make sure the code at least compiles before you try running the test harness; without

compilation nothing is going to work. The key method that does all of the work in the test harness is LOCTest.computeLOC(icu). You do _NOT_ need to modify this in any way.

By default, simply resolving the compilation errors may not result in code that works in the test harness. However, the compilation errors that remain _should_ give you a hint as to what needs to be done to get the code to work. Do not take more than 1 hour to do the task, if you can't finish that's alright, just write about it in the questionnaire; let me know what worked, what didn't, and what your thoughts are about why it did/didn't work.

Again, success for this task is a) resolving all of the compilation errors, and b) pressing the 'Test LOC Counter' button computing lines of code in the console view of the workbench. When you're done, press the Save plan button in Gilligan.

Task 2: Reusing the network throughput view from Azureus

The goal of this task is to reuse the line-graph view in Azureus that is used to visualize its network throughput. The developer wants to use this view in their own application. The reuse plan for this task has been included in the task "Network View - azureus in azureus_target". Your source project for this task azureus; the project you want to reused the code in is azureus_target.

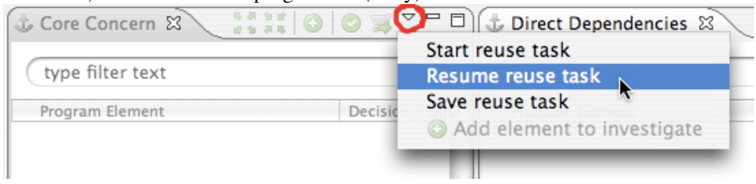
The target project contains a couple of utility classes: TheLog for logging functionality, and MyColours for keeping track of the target application's colour scheme. In addition, there is a test harness (in the form of an RCP application) in the test package. The main part of the test suite that interacts with the reused code is in test.View.createPartControl(parent). To run this test suite, perform the reuse task and ensure the code compiles. You should not have to alter the test code at all. Open the Manifest.mf file in the META-INF folder of the target project. On the overview page tab (bottom left of the editor) is a 'Testing' sub-section in which you can click 'Launch an eclipse application in debug-mode'. This will start up an RCP application which should display a line chart with a whole bunch of random data in it.

The success criteria for this task is a) resolving all of the compilation errors in the reused source code, and b) running the provided RCP application and seeing a randomly-generated line graph in its view. Do not take more than 1 hour to do the task, if you can't finish that's alright, just write about it in the questionnaire; let me know what worked, what didn't, and what your thoughts are about why it did/didn't work.

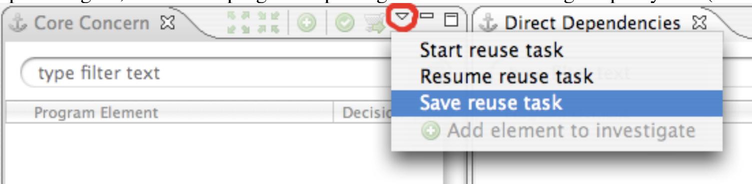
The Treatments

When I emailed you I specified which treatment you should use for each task; these two treatments provide two alternative approaches. Both treatments work from a reuse plan described using the Gilligan tool. This plan describes the classes, fields, and methods that will be reused and how their dependencies should be managed. For details, see the Tools section. The automated treatment uses the Procrustes tool to automatically enact the provided reuse plan. The manual treatment has you use the tools you would normally use in your day-to-day programming activities to enact the provided plan. Please make sure you use right the treatment for the right task (and that you don't accidentally use Procrustes for the manual task).

To start either treatment, load up the provided reuse plan by pressing the caret menu in the Core Concern View and selecting 'Resume reuse task'. Select the task you are performing for this treatment from the dialog and press the 'Resume task' button; the reuse plan will then load (this can take a while (but less than 5 minutes) and has minimal progress UI (sorry).

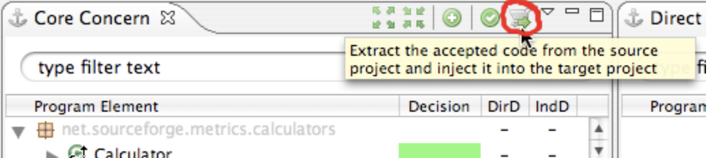


When you are finished with either treatment, save the reuse plan again (even though you would have not modified it). This stops the clock and writes the log files to disk with a time-stamp so I know how long the task took. To do this, press the caret menu in the Core Concern View and select the 'Save reuse task' option. Again, there is no progress reporting for this view but it goes pretty fast (30 seconds or less).



Automated enactment

The automated treatment involves using Procrustes to enact the provided reuse plan. Once you have loaded the reuse plan (described in treatments), you can select the 'enact plan' (shopping cart) button in the Core Concern View toolbar.



If you are performing the first task using this treatment you should have 3 errors to fix in the reusedCode source folder in the target project; if you are performing the second, there should be 11. In addition to resolving the compilation errors you need to make sure the code runs with the provided test harness (see task descriptions). Often the compilation errors that exist can provide clues to help make the code function correctly. In addition, Procrustes also makes simple mistakes which you can easily manually fix.

Procrustes will then enact the reuse plan, copying all of the required code to the target project, modifying it as necessary. Procrustes is a light-weight tool and makes mistakes. Your goal for this treatment is to resolve any compilation errors (not warnings) that still remain in the target project after running Procrustes and make the reused code work with the test harness. All of the code will be copied into the 'reusedCode' source folder within the target project. You don't particularly need to worry about the reuse plan for this task, just go through the remaining errors and fix them so the code compiles and make any other modifications you think are necessary to get the test harness to work.

When you are done, save the task (described in treatments). Please try to do the treatment in its entirety from when you first load the reuse plan until saving the plan, without getting distracted.

Manual enactment

The manual treatment involves manually copying the classes to be reused into the target system and modifying them according to the reuse plan. In this treatment it is important that you follow the reuse plan as closely as possible. At the end of the task you should have no compilation errors remaining and the test harness should work as described in the tasks section. Once you have loaded the reuse plan (described in treatments) you can start using Gilligan to figure out which classes to reuse. Copy these classes (using cut and paste) from the source project to the target project. Make _sure_ you use the same package structure within the target project as the source project. Once you have copied the classes, remove the rejected fields and methods from these classes and references to these fields and methods. Next, remove rejected references to fields and methods from classes that were not reused. Finally, satisfy any remapped, extracted, or injected decisions within the reuse task (what this all means is described in the Tools section. This treatment will involve a lot of switching between the Gilligan perspective and the Java perspective. If you are performing the first task (metrics) using this treatment you should have copied 10 classes to the target

project; if you are performing the second (azureus), you should have copied 6.

For this treatment it's really important that you understand how the annotations in the plan correspond to what you're supposed to do in the source code. That's why you performed the automated task first so you can see how Procrustes modified the source. The detailed description of how to handle each annotation is in the list of annotations in the Gilligan section.

For more information on what's in the plan (especially for remapped entities), you can look at the textual description of the plan itself. This can be found at {workspace location}/.metadata/.plugins/ca.ucalgary.cpsc.skipper.ui/ in the appropriate .gil (xml) file. The decorated section, (near the top in the gilligan->state->decorated sub-tree) lists all of the classes, methods, and fields that have been annotated, and in which way. While this stinks, it may help you overcome any UI issues you may encounter with Gilligan.

The treatment is complete when you feel that you have enacted the reuse plan to your satisfaction, have resolved any compilation errors (not warnings) in the target system, and the test harness functions as described in the task description.

When you are done, save the task (described in treatments). Please try to do the treatment in its entirety from when you first load the reuse plan until saving the plan, without getting distracted. Also, make sure you do _not_ press the 'enact plan' button in Gilligan.

The Tools

Gilligan is a system for planning source-code reuse tasks in a lightweight manner. It consists of a simple UI that allows you to navigate the dependencies in the source code you wish to reuse and mark-up the source code with annotations that describe how you would like the code to be reused. Procrustes is an extension to Gilligan that takes a Gilligan reuse plan and automatically extracts the source code to be reused and integrates it with your project; during this process Procrustes resolves as many compilation errors as possible. Used together, Gilligan and Procrustes provides a platform to quickly prototype reuse activities. The tools enable you to try out different reuse plans and quickly see how the alternatives look in the code by enacting them and examining the remaining compilation problems.

Gilligan Reuse Planning Environment

Gilligan has two primary goals: 1) to help you investigate the dependencies in some source code you are interested in reusing in a lightweight manner; and 2) to annotate that source code with a simple description of how the reuse task should take place. The end-product of using Gilligan is a reuse plan that describes which classes, methods, and fields you want to reuse, and how the dependencies they have on the rest of their system should be dealt with during the reuse task. In this evaluation, the plans are already made so you'll just be using the tool to navigate the annotations you need to follow to perform the manual treatment.

The Gilligan interaction mechanism has been set-up to make it easy to see the direct and indirect dependencies of any source code element at the method/field level. When an element is in the core concern view, by simply selecting it the direct dependencies view will be populated with those elements your selected element is dependent on. You can then either select all of the elements in the direct dependency view or click on the element in the core concern view again to see the direct and indirect (transitive closure) dependencies of your selected elements. If you have a class in the core concern view, you can see its child methods and fields by right-clicking on it and selecting 'Show Children'. You can also move elements from the direct and indirect views to the core concern by right clicking on the element and choosing 'Move to core concern'. Note: the selection behaviour is a little finicky, try not to get too frustrated with it. Most of the key classes for each treatment have been added to the concern view on the left. It is important to note that Gilligan _only_ displays dependencies for methods and fields, and then only for those that have been selected. As such, often a class in the direct dependencies view won't be showing all of its children. This can be seen by the little black 'c' at the bottom left of the class/interface icon (as seen here:).

The <init> and <clinit> methods represent constructors and static-constructors (aka the field-allocation etc) respectively.

While exploring dependencies may be interesting, Gilligan is able to record your decisions about particular dependencies; these are displayed using various colours in the decision column in each view. The decisions you can make are listed below:

- **Accept (green):** Any piece of source code you want to reuse must be marked as accepted. If you want to reuse a method or field you must also mark its containing class as accepted (or else it wouldn't have anywhere to go). The one exception to this is extracted fields (see below).
  - For the manual treatment: copy any accepted class into the target project (with the same package structure). All of the other annotations will be changes to these copied elements.
- **Reject (red):** Rejected code are nodes you specifically do not want to reuse. You only need to reject nodes that are depended upon by accepted nodes. e.g. if your accepted method foo() uses bar() but you don't want to reuse bar(), rejecting it will remove references (by commenting them out) from foo() (or any other reused node) to bar().
  - For the manual treatment: Remove or comment out any rejected methods and fields from the accepted classes. Also, any references to fields, or calls to methods from accepted methods to rejected fields/methods should be removed/commented out. If a rejected method call is part of a call-chain you can remove the whole chain (this is what Procrustes does). If a rejected method call is part of a looping guard, you can remove the whole loop.
- **Remap (blue):** You can remap field references and method calls to elements within your own system. For example, you can remap references in the reused code to a field for its definition of the colour red to your own. Alternatively, you can remap calls from one method (say SWT.error(String) to your Logger.error(String)). Remapping fields works best if they are the same type, remapping methods works best if the parameter lists are the same.
  - For the manual treatment: remaps often have two parts: a target method/field and an optional element on which it is called. e.g. to remap error (in SWT) to error (in Logger) you need to specify the optional element (Logger) or else the error call will think it's a publicly available method that doesn't need a prefix (Logger). To see the annotation, right click on the field/method being remapped and choose mark->remap->remap to existing target; the dialog has two parts: the top part specifies the method/field the selected field should be remapped to. The bottom part specifies the optional part (e.g. _log for the field _log (as in _log.error()) or Logger for a static reference (as in Logger.error())). Make sure you press ok and not cancel afterwards so you do not delete the annotation.
- **Remap - extract field (blue with 'ext' key word):** If you want to reuse a field but not it's containing class, you can extract it from its class to another one you are reusing. This is especially handy if you just want to reuse a few constant fields from a large class or interface.
  - For the manual treatment: This just means: pull the method from the class it was declared into the class specified in the plan. You will also have to update any references from the old field location to the new location. To see the annotation right click on the field being extracted and choose mark->remap->extract; you can then see what the target for the remapping is. Make sure you press ok and not cancel afterwards so you do not delete the annotation.
- **Inject code (Inj keyword):** Code can only be injected on reused classes. You can inject any number of methods or fields on a type; these injections are put in the class right after its declaration. Injecting code is particularly effective when combined with method remapping. For example (if you're using Log4J), you can inject a field Logger _log = Logger.getLogger(this.getClass()) and remap any method call from the reused code's error-handling code (say SWT.error(String)) to _log.error(String).
  - For the manual treatment: Just create the methods/fields as specified by the annotation box. To see what should be injected, right click on the class showing the 'inj' decision annotation and choose mark->inject->inject code; the dialog will show what to do. Make sure you select ok and not cancel afterwards so you do not delete the annotation.
- **Already provided (yellow):** Any common elements between the source and target system (such as java.lang.String) are automatically annotated as already provided.
  - For the manual treatment: you don't have to do anything for this annotation.

You can also view an annotated version of the source code; for any node in the Gilligan views, right click and select 'show source'. This will pop up an editor view that has been marked up according to the decisions you have made. While this view is not perfect it gives a quick overview of which decisions remain to be made and can be helpful to make the decisions you have made seem less abstract.

Procrustes Reuse Plan Enactor

Procrustes executes reuse plans created within Gilligan. Procrustes locates all of the accepted classes and interfaces in the reuse plan and copies them to the target system within a source folder called ReusedCode. It then modifies these reused elements according to the plan. Procrustes removes references to rejected fields, calls to rejected methods, and updates calls and references for remapped fields and methods. By automating the enactment of the reuse plan, Procrustes enables an iterative planning process. The developer can change the plan and just click a single button to see what the effects of this change are on the source code. You just won drinks at the grad lounge; let me know, or I'll have to give it to the students in the lab.

If you decide to try out your own tasks you may choose to iterate on your reuse plan. Procrustes currently has a small bug; if you wish to iterate on a reuse plan you must manually delete the ReusedCode source folder in the target project first; inconsistencies can arise if this step is not taken. Note: as only reused code is stored in this folder, your code is in **NO** danger of being deleted or overwritten.

If you do decide to undertake your own tasks, here are some tips to help make your experience more effective: As mentioned previously, Procrustes enables you to quickly investigate alternative reuse plans. While Gilligan can be used to identify relevant structural dependencies, it may not be clear from a reuse plan what the concrete ramifications of a particular reuse decision are in the source code. To see these ramifications, simply change your plan and press the button to enact the reuse plan. You can then check out the reusedCode folder in the target project and see how it varies from plan to plan. One particularly handy way to do this is to look at the compilation errors that remain after the plan has been enacted: will these be easy or hard to fix? how long might they take? might some other decision in my plan resolve many of these errors and make my job easier? Small changes in the reuse plan can have dramatic effects on the source code when the plan is enacted; test alternative decisions, see what works best for the task you are investigating.

Questionnaire

These are open-ended questions and are very valuable to me; please take the time to answer as completely as possible.

1. How good of a job do you think you did during the manual enactment of the reuse plan?
    - Do you have any other comments about this task (what was easy, what was hard, etc.)?
2. How good of a job do you think you did during the automated enactment of the reuse plan?
    - Do you have any other comments about this task (what was easy, what was hard, etc.)?
3. Do you have any other comments relating to the tasks/treatments you undertook?
4. Was there any difference between resolving the compilation errors and making the changes so test harness work?
5. What was different about performing the reuse task manually vs. automatically?
6. Does Procrustes change the utility of Gilligan's reuse plans?
7. Do you have any other comments about the Gilligan/Procustes approach?
8. If you tried any of your own tasks with Gilligan/Procrustes, can you describe them and do you have any comments?

If you have any low-level tool complaints or suggestions, please give them here:

Submitting Your Results

When you're done I'd like you to submit three things:

1. Your questionnaire answers. Please be as verbose as possible! I can call you on the phone as well, if that makes it easier.
2. A zip of the Gilligan log files. These can be found {workspace folder}/.metadata/.plugins/ca.ucalgary.cpsc.skipper.ui/log/ . In this folder you'll see a bunch of 'traceEvent' files. Since it's hard for me to tell you which ones to send, just zip them up and send them all. If you really want to just send two, I'd like the one that was written right after you saved the plan when you finished the first task and the one that was written right after you saved the plan when you finished the second task.
3. A zip of the two target workspaces from your tasks. These can be found at {workspace folder}/azureus_target and {workspace folder}/net.sourceforge.metrics_target. Just zip up the whole directories.

Please email them to me at "rtholmes@cpsc.ucalgary.ca". Please send it to me by Aug 5.

Appendix F

Full evaluation

This appendix contains all of the supporting materials for the evaluation detailed in Chapter 9. The test harnesses provided to the participants for their tasks are each included (Figure F.3, Figure F.4, and Figure F.5). The test data file used in the QIFParser task is also included (Section F.1).

Four questionnaires were used in this experiment. The entrance questionnaire is included in Section F.2 (p. 187). The mid-task questionnaire, which was answered by the participants up to 9 times is given in Section F.3 (p. 189). The post-task questionnaire which was administered after each task (3 times in total) is provided in Section F.4 (p. 191). The final questionnaire is given in Section F.5 (p. 193).

Our pragmatic reuse plans for the QIFParser task (Figure F.1) and the RelatedArtists task (Figure F.2) from the first experiment are also included.

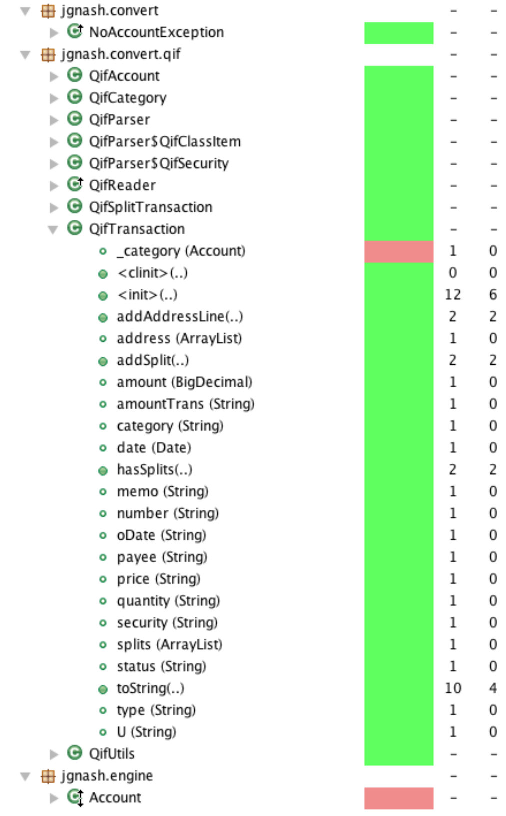


Figure F.1: Pragmatic reuse plan for the QIF Parser task. Common dependencies have been elided to save space.

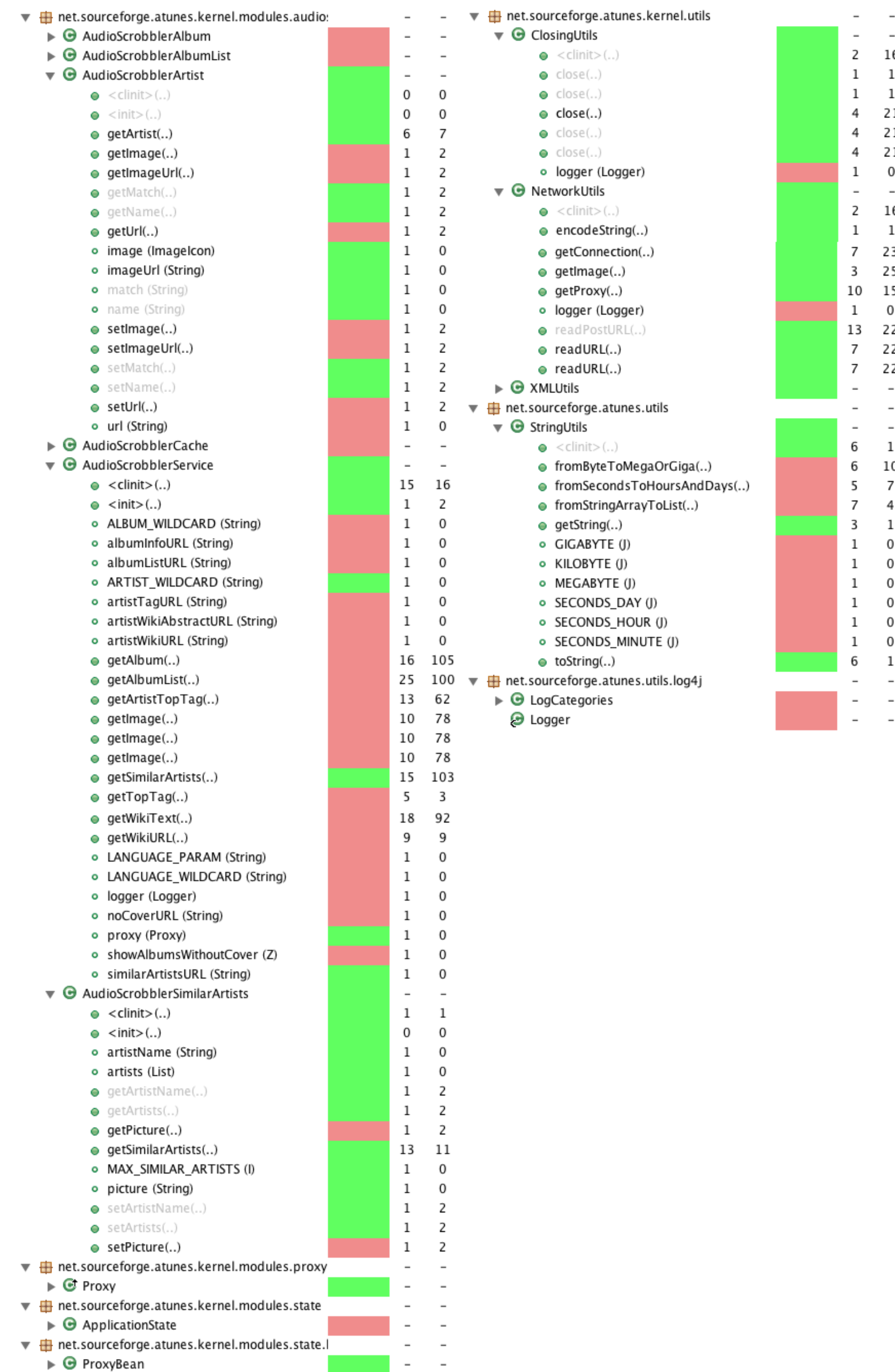


Figure F.2: Pragmatic reuse plan for the Related Artists task. Common dependencies have been elided to save space.

```java
package test;

import java.io.File;

public class QIFTest {

    @Test
    public void testQIFParse1() {
        try {
            QifParser parser = new QifParser(QifUtils.US_FORMAT);
            parser.parseFullFile(new File("ExportNoPrices2.qif"));

            Assert.assertTrue("Sample file contains 39 securities",
                    parser.securities.size() == 39);
            Assert.assertTrue("Sample file contains 1 account",
                    parser.accountList.size() == 1);
            Assert.assertTrue("Sample file contains 0 classes",
                    parser.classes.size() == 0);
            Assert.assertTrue("Sample file contains 0 categories",
                    parser.categories.size() == 0);
        } catch (NoAccountException nae) {
            Assert.fail(nae.getMessage());
        }
    }

    @Test
    public void testQIFParse2() {
        try {
            QifParser parser = new QifParser(QifUtils.US_FORMAT);
            parser.parseFullFile(new File("Export.qif"));

            Assert.assertTrue("Sample file contains 39 securities",
                    parser.securities.size() == 39);
            Assert.assertTrue("Sample file contains 2 account",
                    parser.accountList.size() == 2);
            Assert.assertTrue("Sample file contains 0 classes",
                    parser.classes.size() == 0);
            Assert.assertTrue("Sample file contains 0 categories",
                    parser.categories.size() == 0);
        } catch (NoAccountException nae) {
            Assert.fail(nae.getMessage());
        }
    }
}
```

Figure F.3: Test harness for the QIF parser task.

```java
package test;

import junit.framework.Assert;

public class RelatedArtistTest {

    @Test
    public void getRelatedArtists() {
        try {
            String aName = "Dave Matthews";

            AudioScrobblerService ass = new AudioScrobblerService(null);
            AudioScrobblerSimilarArtists assa = ass.getSimilarArtists(aName);

            Assert.assertFalse("Related artists not retrieved", assa == null ||
                    assa.getArtists() == null || assa.getArtists().size() < 1);

            System.out.println("Artists related to: " + aName);
            for (AudioScrobblerArtist asa : assa.getArtists()) {
                System.out.println("\t" + asa.getName());
            }
        } catch (Exception e) {
            // this should never happen
            Assert.assertTrue("Unexpected exception thrown: " +
                    e.getMessage(), false);
        }

    }

}
```

Figure F.4: Test harness for the related artists task.

```java
package test;

import junit.framework.Assert;

public class TorrentDownloaderTest {

    @Test
    public void testTD() {
        try {
            TorrentDownloaderImpl tdi = new TorrentDownloaderImpl();
            tdi.init(new TorrentDownloaderManager(),
                    "http://www.mininova.org/get/1020084", "",
                    "outFile.txt");

            tdi.run();
            tdi.join();

            Assert.assertTrue((tdi.getDownloadState() ==
                TorrentDownloader.STATE_DOWNLOADING ||
                tdi.getDownloadState() ==
                    TorrentDownloader.STATE_FINISHED));

            Assert.assertNotSame(tdi.getDownloadState(),
                    TorrentDownloader.STATE_ERROR);

        } catch (Exception e) {
            Assert.fail("This should not happen: " + e.getMessage());
        }
    }
}
```

Figure F.5: Test harness for the torrent downloader task.

```
!Option:AutoSwitch
!Account
NZRet. - Fidelity - Mutual Funds
TPort
^
!Clear:AutoSwitch
!Type:Security
NMicrosoft Corp
SMSFT
TStock
^
!Type:Security
NFannie Mae
SFNM
TStock
^
!Type:Security
NFidelity National Financial
SFNF
TStock
^
!Type:Security
NJohnson & Johnson
SJNJ
TStock
^
!Type:Security
NPaycheck
SPAYX
TStock
^
!Type:Security
NSLM
SSLM
TStock
^
!Type:Security
NSysco
SSYY
TStock
^
!Type:Security
NCoca Cola Co
SKO
TStock
^
!Type:Security
NUnilever N V
SUN
TStock
^
!Type:Security
NColgate-Palmolive
SCL
TStock
^
!Type:Security
NNeenah Paper Inc
SNP
TStock
^
!Type:Security
NClipper
SCFIMX
TMutual Fund
^
!Type:Security
NExcelsior Value & Restructuring
SUMBIX
TMutual Fund
^
!Type:Security
NKeeley Small Cap Value Fd Inc
SKSCVX
TMutual Fund
^
!Type:Security
NThird Avenue Value
STAVFX
TMutual Fund
^
!Type:Security
NThird Avenue Intl Value
STAVIX
TMutual Fund
^
!Type:Security
NWeitz Partners Value
SWPVLX
TMutual Fund
^
!Type:Security
NMuhlenkamp Fund
SMUHLX
TMutual Fund
^
!Type:Security
NBerkshire Hathaway Cl B
SBRK-B
TStock
^
!Type:Security
NTweedy Browne Global Value Fund
STBGVX
TMutual Fund
^
!Type:Security
NBridgeway Ultra-Small Co Tax Advant
```

SBRSIX
TMutual Fund
^
!Type:Security
NThird Avenue Real Estate Value
STAREX
TMutual Fund
^
!Type:Security
NBridgeway Balanced Portfolio
SBRBPX
TMutual Fund
^
!Type:Security
NDelafield Fund Inc
SDEFIX
TMutual Fund
^
!Type:Security
NGeneral Amer Invstrs Co
SGAM
TMutual Fund
^
!Type:Security
NBridgeway Aggressive Investor 2
SBRAIX
TMutual Fund
^
!Type:Security
NDriehaus Emerging Markets
SDREGX
TMutual Fund
^
!Type:Security
NIcap International Fund
SICEUX
TMutual Fund
^
!Type:Security
NPerritt Emerging Opportunities Fund
SPREOX
TMutual Fund
^
!Type:Security
NExcelsior Real Estate
SUMREX
TMutual Fund
^
!Type:Security
NGamco Gold Fund
SGOLDX
TMutual Fund
^
!Type:Security
NIshares Inc Msci Switzerland Index Fd
SEWL
TOther
^
!Type:Security
NSsga International Stock Selection
SSSAIX
TMutual Fund
^
!Type:Security
NEBAY INC
SEBAY
TStock
^
!Type:Security
NIshares Inc Msci Mexico Free Index Fd
SEWW
TOther
^
!Type:Security
NVanguard Sector Index Fds Vanguard
Telecommunic
SVOX
TOther
^
!Type:Security
NIshares Inc Msci Singapore Index Fd
SEWS
TOther
^
!Type:Security
NISHARES INC MSCI BRAZIL FREE INDEX
FUND
SEWZ
TOther
^
!Type:Security
NJANUS CONTRARIAN FUND
SJSVAX
TMutual Fund
^

Pre-Study Questionnaire

Participant ID:

Current occupation?
- Undergraduate Student
- Graduate Student
- Industrial Developer

If you are a student, have you ever been employed as a developer in industry?
- Yes
- No

If yes, for how long?

Years of development experience (basic, pascal et al. do not count)?

Years developing with Java?

How familiar are you with Java?
- 1 - Not familiar, have never used it
- 2 - Have used it once or twice
- 3 - I have used it for a short periods in the past
- 4 - I often use it, or have actively used it in the past
- 5 - It is my everyday development language

Do you regularly use an IDE while developing software?
- Yes
- No

If yes, which IDE do you primarily use?

Have you ever used the Eclipse IDE?

How familiar are you with Eclipse?
- 1 - Not familiar, have never used it
- 2 - Have used it once or twice
- 3 - I have used it for a short periods in the past
- 4 - I often use it, or have actively used it in the past
- 5 - It is my everyday development environment

Unanticipated Reuse Study - Spring 2008

Have you ever reused code using copy and paste?

- Yes
- No

If yes, how often would you say you reuse code this way?

What three adjectives would you use to describe the benefits of these reuse tasks?

1)
2)
3)

What three adjectives would you used to describe impediments to these reuse tasks?

1)
2)
3)

Unanticipated Reuse Study - Spring 2008

Mid-Task Questionnaire

Participant ID:

Task:

- Reuse QIF parser from jGnash
- Reuse related artists feature from aTunes
- Reuse TorrentDownloader from Azureus

Treatment:

- Tool-supported
- Manual

Time interval:

- 5 minutes
- 15 minutes
- 40 minutes

Task complete:

- Yes
- No

On the right track:

- Yes
- No

This should take < 60 seconds

I am making good progress:

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

I will be able to complete this task:

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

How much longer do you think it will take:

1) Less than 5 minutes
2) Between 5 and 10 minutes
3) Between 10 and 20 minutes
4) Over 20 minutes

Unanticipated Reuse Study - Spring 2008

What is the #1 problem you are currently having:

What is the #1 thing that is going right:

Any other comments?

Notes

Unanticipated Reuse Study - Spring 2008

Post-Task Questionnaire

Participant ID:

Task:
- Reuse QIF parser from jGnash
- Reuse related artists feature from aTunes
- Reuse TorrentDownloader from Azureus

Treatment:
- Tool-supported
- Manual

Time taken:
- Full 40 minutes
- Other:

Task complete:
- Yes
- No

Close:
- Yes
- Medium
- No

How did you find the task:
- 1) Difficult
- 2) Moderately Difficult
- 3) Okay
- 4) Moderately Easy
- 5) Easy

Did the task take more time or less than you anticipated at the beginning:
- More
- Less

If you needed this functionality in your own line of work, would you perform this reuse task?
- Yes
- No

Why, or why not?

Unanticipated Reuse Study - Spring 2008

What three things made the task hard?

1)

2)

3)

What three things made the task easy?

1)

2)

3)

If performing the tool-assisted treatment:

Did Gilligan help you perform this task?

Did Gilligan hinder you in performing this task?

What did you like about Gilligan, in this case?

What did you dislike about Gilligan, in this case?

Did you find the overhead in creating the plan overwhelming?

Did automating the enactment of the plan helpful?

Was validating your reuse plan useful?

Did you iterate on your plan? Was this helpful?

Do you have any other comments?

Exit Questionnaire

Participant ID:

Manual Tool-Supported

Reuse QIF parser from jGnash
Reuse related artists feature from aTunes
Reuse TorrentDownloader from Azureus

Compare and contrast your experience between the manual and tool-supported approaches:

List three concrete differences between the tool-supported approach compared to the manual approach:

1)

2)

3)

Why those three?

List three concrete differences between the manual-approach compared to the tool-supported approach:

1)

2)

3)

Why those three?

Were the tool-supported tasks easier or harder to perform? Why?

The overhead required to use Gilligan was too much:

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

Unanticipated Reuse Study - Spring 2008

Using Gilligan, I was better-able to understand the scope and implications of my reuse task?

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

Using Gilligan, I was more likely to succeed at an unanticipated reuse task?

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

I feel I could attempt larger reuse tasks using Gilligan than if you were to perform them manually?

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

Do you have any other comments?

Unanticipated Reuse Study - Spring 2008

Appendix G

Final Experiment: Card Sort Data

During the final experiment 254 pages of hand-written notes were taken while the participants performed their experimental tasks; these notes primarily comprised of statements made by the participants but also included some observations. The notes were transcribed onto 90 type-written pages that were split into individual comments, thoughts, or actions. The resulting 954 comments and observations were analyzed using a grounded theory approach [Corbin and Strauss, 1990]. Grounded theory allowed us to group the comments according to their content using some coding criterion. We used an an open coding approach [Miles and Huberman, 1994] to assign codes to the collected data; by not pre-defining our coding strategy we allowed the categories to be iteratively developed and refined. The groups were identified by iterating on the comments four times. After the first iteration, 57 individual themes were delineated that contained between two and 60 comments. The second iteration shrunk the number of themes to 44 but further subdivided these into 136 sub-themes; during this process many sub-themes were merged and split as further commonalities and divergences were found. During the final phase, the 44 themes were grouped into 8 individual concept categories; each of these represented a high-level concept that unified its constituent themes. These concept categories comprised between 3 and 9 themes.

Five of these concept categories were *organic*— that is, they naturally arose from the data; the developers were not answering specific questions that were asked of them. These concept categories each pertained to a different aspect of exploring, analyzing, and performing a pragmatic reuse task. Three of the concept categories (including 16 themes and 66 sub-themes) were *prompted* as they were a result of both observation and participant answers to specific questions that were asked of them. Section 9.7.1 describes the organic concept categories while Section 9.7.3 describes the prompted concept categories.

An overview is provided for each concept category, along with its themes and sub-themes. For each theme the total number of quotes given and the number of individual participants who gave them is enumerated as an indicator of the support for that theme. A synthetic quote is provided for each theme; the intent of this quote is to give an general understanding of what that theme represents.

These synthetic quotes were generated by combining portions of the collected quotes and integrating them into a single cohesive statement. A number of significant, interesting, or demonstrative quotes are also included to further re-enforce each sub-theme.

G.1 Organic Concept Categories

The five organic concept categories that arose from the card sort were both surprising and interesting. They they do not map directly to the steps developers undertake while performing pragmatic reuse tasks, rather they comprise of a combination of actions and mental processes that developers must perform and consider while considering these tasks. Each of the concept categories is intertwined; a short overview of each of them is given here, along with a quick discussion of how they are related. As the concept categories were derived from the data, not fit into pre-defined bins, they do not fit together in a way that supports a natural, linear progression through them. They are presented in an order that tries to be as natural as possible for the reader; however, forward referencing is unavoidable and each of these categories takes place concurrently and iteratively as the developer progresses throughout their reuse task.

The high-level concept categories correspond to dependency identification (Section G.1.1), understanding (Section G.1.2), maintaining mental models (Section G.1.3), hypothesis testing (Section G.1.4) and performing pragmatic reuse tasks (Section G.1.5). These categories can be further grouped into three groups: those pertaining to actions, those pertaining to understanding, and those that are a hybrid. The dependency identification and performing categories both involved the participant commenting on actions they were making to move throughout the source code. The understanding and mental models categories focused on the participant building an accurate representation of the source code so they could be effective in their task. The hypothesis testing category is a hybrid of these other categories: the participants were actively performing small-scale tasks to further their understanding of what they were doing.

Dependency identification. Dependency identification is a specialized form of information gathering. Participants in this phase were trying to identify what structural dependencies were in the system and where the code representing these dependencies existed. They would use the dependencies they found to build their understanding of the source code and translate this understanding into their mental model of how the system was structured and functioned. The methodologies the participants used to identify dependencies remarkably different depending on the experimental treatment they were using for each task; in the manual treatment the participants spent most of their time poring over the source code in a line-by-line fashion, while in the Gilligan treatment the developers quickly navigated throughout the system.

Understanding. The participants built their understanding of how the system worked and was structured by investigating the dependencies and the source code within the system. The treatments did not diverge very much for how the participants built this understanding; in both cases the participants would try to rely on the naming of the structural elements as much as possible to infer some element's functionality to avoid having to read the code in an in depth manner. With both treatments the participants would sometimes need to resort to looking very carefully at the individual lines of source code; sometimes the names alone were not enough. The need to understand would be prompted by the developer identifying a dependency that seemed interesting and would be persisted by incorporating the dependency into the participants mental model of the reuse task.

Mental models. Building an effective mental model was of critical importance for the participants to be successful in their reuse task; without a model of how the code the participant wanted to reuse worked they lost track of the details and ended up following false paths, getting lost, and making more work for themselves. Ultimately, Gilligan provided far greater support for encoding the decisions the participants made while building up their understanding of the task and consequently the participants felt like they were much better able to focus on the task without being distracted while performing the Gilligan-supported treatments.

Hypothesis testing. Performing a pragmatic reuse task is a heavily iterative process. One of the main reasons for this iterative process was in the pursuit of hypothesis testing. Once a participant had determined that a structural element was relevant to their reuse task they would attempt to reuse it by coping it to their system. They would then analyze the resulting errors and decide whether the errors induced by reusing the code outweighed the benefits of reusing the structural element or not. The results of this process both built their understanding of the system and contributed to their mental model. The treatment the participants were employing had a huge effect on the participant's willingness to investigate different hypotheses; manually this process was very difficult to do in an in-depth manner, while with Gilligan the participants could easily change their mind if necessary. While this could be viewed as a specialization of performing the pragmatic reuse task, its emphasis on developing the participant's understanding and building their mental model makes this category significant and independent.

Performing pragmatic reuse tasks. Themes related strictly to performing pragmatic reuse tasks fell into this concept category. Of the five concept categories this is the least surprising. These themes concern strategies and methodologies used by the participants in both treatments to turn their mental model into a complete pragmatic reuse task. The approaches used by the participants for the treatments are unsurprisingly divergent, as Gilligan automated the majority of this work for the participant; however, Gilligan's support for rejecting structural elements had an impact on the participants

and how they understood and built their mental models for their tasks.

Table G.1 lists the 25 themes encapsulated by the 5 concept categories. Each theme is listed in the order they appear in the subsequent text. The table also reports the number of participants who made quotes that fell into each theme, the total number of quotes in the theme, and the theme's synthetic quote.

# Participants	# Quotes	Synthetic Quote
Dependency Identification		
16	176	
16	39	*It is frustratingly difficult to manually identify the relevant dependencies for a pragmatic reuse task.* ▶
8	10	*It can be overwhelming and disorienting to manually navigate through unfamiliar source code.* ▶
7	9	*Using Gilligan it is easy to locate and navigate through source code dependencies.* ▶
15	40	*It is important to be able to measure and track progress while performing complex reuse tasks.* ▶
14	29	*Gilligan provides the majority of the information necessary to make informed decisions about reuse plan alternatives.* ▶
14	32	*Gilligan's validation view can help developers avoid spending large amounts of time wading through lengthy lists of compilation errors.* ▶
7	15	*Being systematic helped developers feel that they were better able to manage large, complex tasks.* ▶
2	2	*Gilligan helped developers spend less time investigating irrelevant common dependencies.* ▶

# Participants	# Quotes	Synthetic Quote
Understanding		
16	58	
12	18	*The source code is the authoritative source of information. Gilligan reduces the burden of having to read the code but provides access to it as needed.* ▶
6	11	*Understanding the functionality provided by the source code is key to determining whether it should be considered for reuse.* ▶
4	5	*Manual investigations tend to be breadth-first while Gilligan enables a more explorative methodology.* ▶
14	24	*Poorly named or modularized source code can obscure its functional role from a developer.* ▶
Mental Models		
14	74	
9	16	*Gilligan explicitly encodes reuse decisions; this reduces overhead required to manually remember all of the salient details.* ▶
11	26	*By providing a high-level view of a reuse task, Gilligan makes it easier for a developer to get a global understanding of the source code they are reusing.* ▶
5	6	*Gilligan helps developers focus on their reuse task rather than being overwhelmed by numerous compilation errors.* ▶
6	7	*Gilligan helps developers manage the complex details that arise during pragmatic reuse tasks.* ▶
8	19	*Developers are encouraged and supported in making explicit, consistent, decisions using Gilligan.* ▶

# Participants	# Quotes	Synthetic Quote
Hypothesis Testing		
16	76	
12	32	*Developers frequently make poor decisions while performing pragmatic reuse tasks; these are difficult to reverse manually.* ▶
11	14	*Gilligan encourages and assists developers in investigating alternative reuse strategies.* ▶
12	25	*By helping developers progress through a pragmatic reuse task one decision at a time, Gilligan enables them to better track their successes and failures.* ▶
3	4	*Developers using Gilligan are more confident in the quality of their solutions.* ▶
Performing Pragmatic Reuse Tasks		
16	91	
14	20	*Most of the manual work required during a pragmatic reuse task is conceptually simple but labour intensive.* ▶
14	41	*Reuse tasks require less time, effort, and frustration when they are performed with Gilligan.* ▶
7	12	*Developers are likely to copy entire packages while manually performing a reuse task, even though they know it is likely to backfire on them.* ▶
8	18	*Being able to easily remove elements that are not related to the reuse task is essential to reusing only relevant source code.* ▶

Table G.1: Overview of the organic concept categories.

G.1.1 Dependency Identification

All 16 participants contributed 176 individual quotes related to identifying relevant structural dependencies. These quotes coalesced into 8 major themes, as shown below. The greatest problems the participants had was with locating all of the dependencies so they could decide what to do about them. Identifying and locating dependencies is one of the most important tasks the participants needed to perform to successfully complete their reuse tasks; before they could understand the implications of reusing a dependency to decide whether or not they would reuse it they had to first figure out what the dependencies were and then locate their representations in the source code. While all of the information necessary to find each dependency can be found in the code, the participants performing the manual treatments had to expend much of their effort on investigative work, which negatively affected the amount of time they had for understanding what they had found or performing the reuse task.

The participants coped with these drawbacks by trying to keep appraised of their progress; this helped them to feel better about their reuse task. One major way of remaining appraised used by both treatments was monitoring the quantity of compilation errors present in the reused source code. Being systematic was another mechanism the participants employed to try to manage the complexity of the reuse task; several of the participants found that Gilligan encouraged them to be even more systematic than they were manually and that this was helpful for them.

Gilligan's information delivery mechanism was quoted by 14 of the participants as a helpful means for locating and traversing dependencies. They felt that Gilligan helped to filter the information so they could focus on the task which enabled them to make more informed decisions about the relative costs of reusing a dependency. Participants also tended to get lost much less frequently while performing the Gilligan treatments as the structural dependency views provided a natural "bread crumb trail" that helped reduce disorientation.

G.1.1.1 Identifying Dependencies: *It is frustratingly difficult to manually identify the relevant dependencies for a pragmatic reuse task.*

All 16 participants made 39 comments about their frustrations with finding the list of structural dependencies for any piece of source code. While the source code does contain all of the details necessary to identify all of the structural dependencies, this information would often be scattered throughout many files (especially when inheritance was involved). For the manual treatments this meant many editor windows would flash open while the participant would investigate any dependency. Particularly, the insidiousness of indirect dependencies to foil the participant's reuse intentions was bemoaned. Often the participant would copy a class or method to their system only to find out that it caused even more errors than they had before. One simple way participants used to get an overview of the dependencies was to look at the import list for a class, although this was terribly inadequate because classes do not list dependent classes in their own package and import lists are

frequently wild-carded (e.g, `java.util.*`).

Manually working through all of the dependencies is a lot of work.

> Manually I had to sort through a lot of information to find what I needed. (P7)
>
> I'm [having trouble] trying to understand the system; there's a lot of stuff to go through. (P11-T3; TD-M)
>
> There are a lot of dependencies that look unrelated to my task but I'm not so sure. The code isn't the prettiest either! (P15-T3; TD-M)

Indirect dependencies are surprising and problematic.

> Resolving the dependencies was hard: something that looked innocuous to reuse proved to be a nightmare. (P1-T3; TD-M)
>
> Every time I think I'm getting close [the # of errors] explodes! (P3-T2; Q-M)
>
> By hand it's very hard to tell by looking at the code what costs are associated with reusing a dependency. (P8)
>
> The dependency cycle is almost endless! You never know the ramifications of reusing a class in your system. (P16-T1; Q-M)

Using the import statements to find relevant dependencies is insufficient.

> Copying the `Debug` class was harder than it looked because package-level dependencies aren't listed in the imports. (P1-T3; TD-M)
>
> Not having fully-qualified import statements is impeding my investigation. (P2-T2; RA-M)

G.1.1.2 Navigation: *It can be overwhelming and disorienting to manually navigate through unfamiliar source code.*

8 participants made 10 comments about the problems they experienced navigating through dependencies in the source system. Scrolling blindness was a problem for participants when they were performing the manual treatment of the tasks as they were continually scrolling through different editors and views and would frequently lose track of their position in the system. This was further

exacerbated as the number of editors they had open increased as they navigated along structural dependencies to new parts of the system. Being *lost* was a major source of confusing for participants. The primary kind of *lost* participants experienced was forgetting how they got to their current location in the code; this meant they had forgotten aspects of their mental model (see Section G.1.3 for more details about mental models) and caused the participants to waste time revisiting previous pieces of code they had visited in an attempt to retrace their steps. Participants infrequently got lost using Gilligan as they could usually just look to the structural view to the left of the one they were using to see the path they had taken; if they were in the left-most view this would not work, but developers in this view were always looking at code they had previously annotated and made concrete decisions about which they used to quickly ground themselves.

I'm doing an awful lot of scrolling here.

> The tool felt easier because I didn't have to scroll through and copy and paste using the package explorer; without all of that manual work it was faster. (P12)

There are lots of things to navigate between.

> I had a lot of files opening up; there were lots of editors to navigate through. (P9-T1; RA-M)
>
> Lots of [switching] back and forth between the source and target systems really interrupted my workflow. (P11-T2; RA-M)

It's easy to get lost in all of the editors I have open.

> The manual approach is more "fly-by-wire" as you hop from thing to thing; it is easy to get lost. (P2)
>
> Often I lost track of the method I was investigating. (P5-T1; RA-M)

G.1.1.3 Gilligan — Finding dependencies: ***Using Gilligan it is easy to locate and navigate through source code dependencies.***

7 participants made 9 comments about the benefits of Gilligan quickly displaying the structural dependencies for source code element. Ultimately, these participants were comparing the Gilligan experience to the frustrations they experienced while trying to locate and navigate through dependencies manually using an editor. In addition to quickly outlining the dependencies, participants used the dependency list to quickly navigate into new parts of the system to see how dependencies spread out throughout the system.

> It's kind of like shopping; you can pick and choose what you want and see the dependencies right away. (P11-T1; Q-G)

> [I liked] that I could just browse through the dependencies instead of using the editor which would have taken forever. (P12-T3; TD-G)

> It was easy to see what dependencies existed and navigate to them directly. (P15-T2; Q-G)

G.1.1.4 Measuring Progress: *It is important to be able to measure and track progress while performing complex reuse tasks.*

15 participants made 40 comments about being able to feel (or not feel) like they were making progress in their reuse task. Participants who did not feel like they were making progress often expressed frustration with the reuse task in general. The main way participants measured progress was by monitoring the number of compilation errors that were left for them to consider. Participants performing Gilligan-assisted treatments also used the number of elements in the Validation view as a measure of their progress on a reuse task. By being able to quickly see progress in their reuse tasks participants using Gilligan were often more positive about their effectiveness on the reuse task and felt more enthusiastic about the quality of the reused code.

I use the error count as a measure of how I'm doing.

> The error count is going down. One class is error free. (P11-T2; RA-M)

> The compilation errors guided me so I could see if I was making progress. (P14-T1; RA-G)

> With the tool errors would decrease, but manually I would keep getting more errors. (P13)

I'm having trouble estimating how I'm doing.

> It's so hard to tell [how I'm doing] at this point because there is so much I still need to go through. (P2-T2; RA-M)

> I cannot estimate how well I'm doing manually; with the tool I was more sure of my scope. (P5)

Gilligan's validation view gave me a sense of progress.

> Validation was very valuable to me; being able to check my progress and how things are going was nice. (P1)
>
> Iterating [on my reuse plan] gives me a sense of what is remaining. If I do it piecemeal I avoid making duplicate decisions and doing extra work. (P5-T2; Q-G)
>
> I dig the validation and checkout views; being able to see progress allows for more positive and negative reenforcement so you know where you're going. (P14-T1; RA-G)

G.1.1.5 Information Delivery: *Gilligan provides the majority of the information necessary to make informed decisions about reuse plan alternatives.*

14 participants made 29 comments about how Gilligan's focus on providing information on structural dependencies, and making this data easily navigable, was key to the utility of the tool-supported approach. By clearly highlighting dependencies, the participants could quickly see what dependencies exist without having to resort to reading through the source code which was perceived to be a slower way to find dependencies. Gilligan's focus on providing quick, flexible dependency navigation allowed participants to focus on the dependency itself, and estimating the costs of reusing it, without having to first copy the code, compile it, and trying to identify the new compilation errors relevant to the reuse code separate from all of those that existed beforehand. In effect, Gilligan could act as an oracle, giving the participant a worst-case scenario about the potential costs of reusing a dependency (it is worst-case because some of the dependencies listed in the view may already have themselves been reused and therefore not really been a reuse burden). Ultimately, Gilligan enabled developers to focus on the information itself, rather than the mechanics of collecting the information.

Highlighting dependencies is useful.

> [Using the dependency views] was quicker than reading the source code. (P4-T2; RA-G)
>
> With the tool I could see the dependencies right away; this gave me the foresight to see if I might need other classes or if I should exclude a class. Manually this just hits you after the fact. (P11)

Gilligan gave me the information I needed.

> [Using Gilligan] all of the information you need is right in front of you. (P10-T1; RA-G)
>
> The conceptual model of the tool is simple and maps to how I think about the code anyways. (P14-T1; RA-G)

> Gilligan gave me the pertinent information I needed to do the task and it did a lot of the work for me automatically. (P7)

Gilligan filtered the information for me, helping me to focus on the task at hand.

> [Gilligan] filters the dependencies so you know what you need to concentrate on. (P4-T2; RA-G)

> [With Gilligan] all I have to do is focus on the dependencies; I don't have to focus on the code. This way I can quickly see how far along in the task I can get. (P5)

Gilligan helped me to estimate the cost associated with reusing a dependency.

> [Using Gilligan] I could see when something would be cheap to reuse. (P7-T2; Q-G)

> The direct and indirect dependency views let me see, in advance, how hard it would be to reuse something. (P8)

> Oh wow, this could be bad. [on looking at the classes in the direct dependency view in the `engine` package]. (P13-T1; Q-G)

The dependency counts helped me to make my decisions.

> Having the number of dependencies listed was good so I at least knew what I was getting into instead of blindly accepting and rejecting things. (P12-T3; TD-G)

> [Using Gilligan] you can actually see what the dependencies are. The number is actually intimidating but looking at the dependencies [in the structural views] you can see the source of them; if lots of them are self-referential ten it might not be a big deal. (P16-T2; RA-G)

G.1.1.6 Gilligan — Validation View: *Developers can avoid spending large amounts of time wading through lengthly lists of compilation errors by using Gilligan's validation view.*

14 participants made 32 comments about the role Gilligan's validation view (see Section 8.2.3 for details about the validation view) had on how they identified and navigated through dependencies. Many of the participants still relied on compilation errors to understand the problems they were having, even with the Gilligan-supported treatments. This was because they were more familiar with the compilation errors than with the validation view and because compilation errors are much more

expressive than the validation view could be. Participants frequently used the compilation error list to choose an error that was frequently occurring; they would then select this element from the validation view because for a significant portion of frequently occurring errors could be attributed to a required structural element that they had not yet triaged. The participant could then navigate directly back into the structural views from the validation view; this loop (errors to validation view to structural views) was employed by every participant on the Gilligan treatments. By following this approach the developer could also avoid having to search through the structural views to find the elements they were interested in. One of the main reasons the participants used the validation view was that instead of bringing them to the source of the *error* (as the compilation view does), the validation view would take them to the most likely source of the *solution*.

Even using Gilligan seeing the compilation errors is still important.

> I'm comfortable dealing with compilation errors. (P3-T1; RA-G)
>
> I mostly used the compilation errors to inform which element to choose from the validation view. (P12-T2; RA-G)

Gilligan points out what I haven't looked at yet.

> The validation view let me quickly see what things I've overlooked. (P2-T1; Q-G)
>
> [The validation view] points out, "you haven't looked at these yet". It also gives you a way of navigating back into the abstraction.

Gilligan points out where I should look next.

> [Validating the plan] was a lightweight way to locate TODO items. (P4-T2; RA-G)
>
> Manually it was too hard to prioritize what to look at next because I was already overwhelmed with problems. (P13)
>
> Unlike the compilation errors that showed me the effect of what my decisions were, the validation view provides feedback on where I should go next. (P14-T1; RA-G)

Gilligan's validation view gives me a link back into the structural representation.

> I did the checkout and validation together. I would look at the state of the code and use the validation view to navigate back into the abstraction at the top. (P7-T2; Q-G)
>
> The validation view helps you along. Manually I have to diagnose problems by looking at all of the compilation errors. With the validation view I just navigate back into the abstraction. (P10)

G.1.1.7 Being Systematic: ***Developers felt they were better able to manage large, complex tasks by performing them systematically.***

7 participants made 15 comments about how being systematic helped them to manage the complexity of their reuse task. The participants frequently would say they 'had a plan' when they felt that things were going well and they felt they were making forward progress. When the participants were faced with a daunting number of errors or dependencies to triage they often commented that being methodical and 'working through the list' was an effective strategy to avoid becoming 'bogged down'. Many participants felt that Gilligan encouraged them to be more systematic and the participants consequently felt more effective on these tasks.

I feel like I have a plan.

> I still have a plan. (P6-T3; TD-G)

Being methodical makes things easier.

> Changing to a more methodical approach [made my task easier]. (P15-T1; RA-M)

Gilligan encourages me to be systematic.

> The tool supported approach allowed me to consistently progress through the source code to complete the reuse task. (P2)
>
> [I liked that] Gilligan helped me to be more systematic [for this task]. (P5-T2; Q-G)

G.1.1.8 Common Dependencies: ***Developers wasted time investigating irrelevant common dependencies while performing their reuse tasks manually.***

2 participants made 2 comments about the importance of not re-investigating code that had already been considered. While only two participants made these remarks, it was often observed that participants undergoing the manual treatment would end up either ignoring or investigating dependencies that were common to between the source and target systems. Participants would also express frustration when they realized they had already investigated dependency they were considering and had forgotten about it.

> [Manually] there are less clues on which dependencies are common; if you don't recognize them you have to spend time thinking about what they are. (P2)
>
> I tend to waste time manually managing things that don't matter. The yellow dependencies keep me from getting confused. (P5-T2; Q-G)

G.1.2 Understanding

All 16 participants contributed 58 individual quotes pertaining to understanding source code. These quotes broke down into four main themes: The role of the source code in a pragmatic reuse task, how developers determined what role specific pieces of code had within the context of their reuse task, strategies for navigating and understanding code, and the impact of the quality of the code's modularization on its understandability and reusability.

Pragmatic reuse tasks usually involve reusing source code not written by the developer performing the reuse task. This imposes a large burden in terms of understanding the source system for the developer performing the reuse task. The experimental participants employed a variety of techniques to develop an sufficient understanding of the code they were reusing to be able to complete their tasks.

None of the participants had ever seen the source code being reused in the experimental tasks; to make up for the lack of familiarity they spent a considerable portion of their time trying to understand the code. In general, the participants did not initially try to determine how the system worked, instead they were more interested in trying to determine what functionality each piece of the system provided so they could determine its relevance to their reuse task. Building this initial understanding is crucial to performing pragmatic reuse tasks in a reasonable amount of time (see Section G.1.5 for details of how the participants performed these tasks) as the developer's first goal in performing these tasks is to identify the relevant code before they can consider how to reuse it. Once some code is determined to be relevant, the participants tried to estimate the cost associated with reusing the code (Section G.1.4 outlines some specific approaches leveraged by developers to gain a firmer understanding of these costs).

G.1.2.1 Role of source code: *The source code is the authoritative source of information. Gilligan reduces the burden of having to read the code but provides access to it as needed.*

12 participants made 18 comments about how they used the source code during their reuse tasks. These comments broke down into 3 main sub-themes. The participants felt that during the manual treatments they felt *closer* to the source code; however, this also caused problems as spending all of the time in the code made it hard to gain a high-level perspective on the task (see Section G.1.3.2 for how participants maintained this overview for Gilligan treatments). Even using Gilligan the participants would look at the source code; this would generally take place when the participant was trying to make a *tricky* decision, or at the end of the task when they were making changes to the code that Gilligan could not do for them.

Manually I feel closer to the code.

> Manually you have a real connection to what the code is actually doing. (P4)

> [Gilligan] tended to abstract away more, but I ended up feeling more detached from the code. (P6)

Reading too much code can be overwhelming.

> It's really slow to read the code; there's a lot of scrolling through multiple editors involved. (P11-T2; RA-M)

Even using Gilligan I need to look at the code sometimes.

> The abstraction can hide things; sometimes I had to go to the code directly. (P8-T3; TD-G)

> Being able to view the annotated source code directly from the structural views [was helpful]. (P9-T2; Q-G)

> Using the tool I didn't miss the fact that I wasn't looking at the source code directly because the abstraction was usually sufficient. (P14)

> With the tool I rarely looked at the code; mainly I looked at the code at the end to make only the tricky decisions. (P10)

G.1.2.2 Relevance of dependencies: *Understanding the functionality provided by the source code is key to determining whether it should be considered for reuse.*

6 participants made 11 comments about the role understanding the code had on their task. The participants were continually trying to understand what each dependency in the source code did so they could then decide whether they should be reusing it or not.

Understanding the code is important.

> I don't understand the code I'm reusing. (P7-T1; RA-M)

> In the manual task I was thinking about what I was importing; with Gilligan I had more of an understanding. (P16)

How does this relate to my task?

> I'm trying to understand what I need and what I can get rid of. (P12-T2; RA-G)

> Trying to understand what each class does and which methods I care about. Is it related or can I reject it? (P2-T1; Q-G)

G.1.2.3 Investigation Strategy: *Manual investigations tend to be breadth-first while Gilligan enables a more explorative methodology.*

4 participants made 5 comments about how their investigation strategy differed between the treatments. Manually the developers explored the code in a breadth-first manner. This meant they would look at a specific class or method in its entirety before continuing their investigation by navigating to a dependency in another file; this was generally to contain the number of editors they had open and keep themselves from getting lost. With Gilligan they tended to quickly travel down dependency chains so they could get a sense whether the top-level method or class they were considering really required the dependency chain they were investigating.

> [Manually] I stayed within a realm; I worked on immediate dependencies before I jumped to the next because I didn't want to get lost. (P2)

> With the tool I work through the dependencies in a depth-first manner to see what is important; manually I traverse them breadth first to get a sense of each dependency before I get lost in its children. (P9)

> Manually it was really a breadth-first process whereas with Gilligan I could jump around more to consider what I wanted without losing that higher Perspective. This is more natural because it allows you to follow your way throughout the code without stopping yourself in fear of getting lost or forgetting where you were. (P14)

G.1.2.4 Impact of code quality: *Poorly named or modularized source code can obscure its functional role from a developer.*

14 participants made 24 comments about the ability of the code's quality to interfere with, or aid, their reuse task. These comments broke down into 5 main sub-themes. Poor naming conventions and a lack of comments made it difficult for the participants to assess the functional role of a structural dependency. This especially affected Gilligan users as they were not as connected to the source code itself; the manual users were more likely to invest the effort to try to understand what a particular piece of code that had a poor name did (see Section G.1.2.1). The package structure affected developer's global overview of how the system they were reusing code from was organized. This tended to affect the manual participants more as they spent more time searching through the originating system looking for dependencies (see Section G.1.1.1). Participants in both treatments found it difficult to reuse poorly-modularized code, although participants in the manual treatment found it more difficult (see Section G.2.3.1).

Poor naming conventions and comments interfere with understanding.

The Process is easy but the code sucks and you can't get around that. There are limited comments and the naming conventions are problematic. (P2-T3; TD-G)

This project has the worst names ever. (T5-T3; TD-M)

The structure of the code isn't helpful. I had to do lots of guessing about the purpose of dependencies. (P2-T3; TD-G)

Good naming conventions aid good decision making.

The naming conventions don't lend themselves to understanding what the code is for. (P8-T3; TD-G)

Since I spent so little of my time looking at the code I'm really reliant on a consistent naming scheme and the code itself being of a reasonable quality. (P14-T1; RA-G)

Package structure can facilitate or deter system-level comprehension.

The class and package layout reenforced how the system was laid out. (P1-T1; Q-G)

The package structure was confusing. (P3-T3; TD-M)

Nobody likes spaghetti code. (4 participants; 6 quotes)

The code kinda sucks. There's like dependencies everywhere; it's just spaghetti code. (P15-T3; TD-M)

The code is just a mess; it's spaghetti code. (P15-T3; TD-G)

It is easier to reuse with well modularized source code.

The code is not modularized well enough to reuse it. (P4-T3; TD-G)

The structure of the code itself just doesn't naturally lend itself to reuse. (P11-T3; TD-M)

Not much thinking required, the code was fairly well modularized. (P8-T1; Q-M)

G.1.3 Mental Models

Of the 16 participants taking part in the experiment, 14 of them made 74 comments about developing, and maintaining, mental models. These quotes can be further subdivided into five main themes: 1) The participant's difficulty when they tried to maintain an accurate mental model for a pragmatic reuse task; 2) How they used the overview perspective provided by Gilligan; 3) The importance of explicit decision making to maintaining a clear mental model; 4) How being focused on the task rather than distracting details makes creating an accurate mental model more achievable; and 5) The participant's ability to better handle complex reuse tasks using Gilligan.

One of the largest problems participants performing the manual treatment had was keeping track of the details of their reuse task. As they identified and navigated throughout the dependencies in the system, they decided on their relevance and added it to their mental mode; unfortunately, the participants frequently either forgot important details, or made decisions that they had previously made about the same structural elements. Sometimes the participants would also recall a decision but be unable to recall their rationale for their initial decision.

The participants remarked on a number of aspects of Gilligan and how it aided their retention of their mental model. By capturing all of the details about a reuse task in one place, Gilligan gives an overview perspective that makes it easy to see what all of the details of the reuse task are at a glance. The tool also ensures that decisions are consistently applied across a reuse task, a common problem for the participants performing the manual treatment. To capture the reuse plan, Gilligan forces the developers to explicitly annotate structural dependencies with their decisions; after adjusting to this requirement, the participants found that this was easier than making piecemeal modifications to the code itself. The abstraction mechanism used by Gilligan enabled the participants to 'step back' from the code, enabling them to both see their decisions in context and to insulate them from the syntatic and mechanical aspects of the reuse task; both of these enabled the participants to better develop their mental model to handle complex pragmatic reuse tasks without being overwhelmed by low-level issues.

G.1.3.1 Tracking task details: *Gilligan explicitly encodes reuse decisions; this reduces overhead required to manually remember all of the salient details.*

9 participants made 16 comments related to the difficulties they had keeping track of all of the details of a pragmatic reuse task in their heads. Only 2 of the 16 participants chose to keep notes on a piece of paper as they proceeded through the reuse task; the other 14 tried to remember all of the details. The difficulty of remembering all of the details was exacerbated by the fact that the participants were reusing code they were not familiar with and did not have a prior mental model of the systems to fall back on. Having a strategy helped some of the participants keep track of the details; the main method used was by trying to clear up all of the compilation errors for a file so they could "forget" about it and not have to worry about it anymore; alternative strategies included only keeping editors

open for files that were important or still needed to be considered in some manner. Many of the participants found that having Gilligan maintain the model for them, so they could only reference it as they needed, enabled them to perform their reuse tasks.

Remembering all of the details of my reuse task is difficult.

> [Manually] I had to do the same workflow, but by navigating all of these editors, AND I had to remember everything for myself!. (P14-T2; Q-M)

> Having to remember all of the details without forgetting something important was almost impossible. (P14)

Gilligan maintains the model so I don't have to.

> I could set aside decisions I wasn't ready to make because it wasn't clear what the right decision was until later and not lose track of them. (P3-T1; RA-G)

> Caring about the details seemed more attainable with Gilligan than [manually]. (P3-T2; Q-M)

> [I liked] that I didn't have to remember all of the decisions I had made. (P5-T2; Q-G)

> You sort of browse through the whole system easier, you don't have to retain the whole model in your head; it lets you bounce around a bit more. (P10)

I have a strategy to keep track of my mental model.

> This strategy [considering the packages alphabetically] is probably not the best way to do it but it's just a way of keeping track. (P13-T3; TD-M)

> I'm only going to keep editors open for files that still have errors. (P13-T3; TD-M)

Where did this dependency come from again?

> Why did I copy this? Where is the dependency that required it? (P9-T1; RA-M)

> What did `AudioScrobblerService` need from the `Kernel` again? (P11-T2; RA-M)

G.1.3.2 Gilligan — Overview Perspective: *By provides a high-level view of a reuse task, Gilligan makes it easier for a developer to get a global understanding of source code they are reusing.*

11 participants made 26 comments about the high-level perspective Gilligan provided them for pragmatic reuse tasks. Whereas manually the participants would could get lost within the numerous source code editors and views they were navigating (Section G.1.1.2) and would have trouble remembering the details of their reuse task (Section G.1.3.1), Gilligan's ability to provide a high-level overview of the reuse task alleviated many of their problems. By working with an abstract representation of the problem, the participants were able to concentrate more on the high-level aspects of the problem while worrying less about low-level issues such as syntax and compilation problems. They could get an overview of all of the decisions they had made by glancing at the decisions panel in the structural dependencies views; if they ever forgot about a specific decision they had made they could quickly refresh their memory here without having to go back to the source code and make the same decision again. By working within Gilligan's abstraction, participants were able to codify all of their decisions in one place, rather than making numerous edits and copies to various source code files; this helped them to ensure their decisions were always consistent and explicitly persisted.

Having a high-level view of the task was valuable.

> [Gilligan] provided a high-level understanding of something you don't want to consider at a low level because it's just huge. (P4-T3; TD-G)

> I was able to work at a higher level without worrying about syntax or other little details. (P10-T1; RA-G)

> [The best aspect of Gilligan was] was the holistic perspective it gave me. The 10,000 foot view to look at a dependency and see if reusing it would be painful without having to do all of the work to discover that fact for myself. (P14-T3; TD-G)

> [Using Gilligan] you could see what the whole problem was and never lose perspective by being down some specific path. (P14)

Working with an abstraction was helpful.

> Having the overview perspective made the tasks more manageable. (P4)

> [Gilligan] gives you more abstraction so you can make it farther [in your task]. (P10)

Gilligan's colour coding helped me see what decisions I had made at a glance.

> The colours made things clear, both in the trees and in the code view. (P3-T1; RA-G)
>
> The decisions I had made were clearly visible which is a huge help for me. (P5-T2; Q-G)
>
> I liked Gilligan's colour coding; it makes it really easy to see what I've done so far. (P14-T1; RA-G)

Reuse plans are beneficial.

> [I liked that] Gilligan forced decisions into the plan instead of making piecemeal code modifications. (P1-T1; Q-G)
>
> The plan encapsulates what works and fails in one place. (P2-T1; Q-G)
>
> Manually I started from the top and copied things; with Gilligan I planned things out more. (P12)

G.1.3.3 Gilligan — Task Focus: *Gilligan helps developers focus on their reuse task rather than being overwhelmed by numerous compilation errors.*

5 participants made 6 comments about how they felt they were better able to concentrate on their task using Gilligan. There were 2 main reasons for this: 1) The participants were able to focus on the dependencies involved in the task, instead of being distracted by the numerous compilations attendant with a reuse-task in progress; 2) Rather than having to think about the mechanics of their reuse task they were able to consider their problem at a conceptual level. The second reason is particularly valuable as Gilligan enabled the participants to think about the dependencies and their relevance to their task without then having to invest a lot of manual effort to check to see if their decision was a good idea or not.

> [Gilligan] allowed me to focus more on the complex reuse issues and to modify the system to fit into my overall reuse task. (P2-T1; Q-G)
>
> Once I made a decision [with Gilligan] I never had to consider it again. I was able to concentrate on making decisions than on compilation errors. (P3)
>
> [Gilligan] allows me to keep focused on the task and provides instantaneous feedback on how the task is proceeding. (P14-T1; RA-G)

Interestingly, 3 participants made 3 opposite comments to these about how they focused while performing the tasks manually. These participants vocalized that they would focus exclusively on the errors, rather than trying to figure out how things worked. They would come back to consider the functionality after the code was compiling.

> [Manually] I started by resolving all of the dependencies instead of focusing on the key functionality. (P5-T1; RA-M)

> [Manually] it is more about fixing the compilation problems and seeing how things compile, rather than worrying about how things work; that comes later. First solve all of the red stuff and see if it works after. (P10-T2; Q-M)

G.1.3.4 Gilligan — Handling Complexity: *Gilligan helps developers manage the complex details that arise during pragmatic reuse tasks.*

6 participants made 7 comments about Gilligan's ability to help them deal with the complexity of their reuse task. Surprisingly, the participants noted that they felt that using the tool helped them to investigate the third reuse task for longer than if they had been attempting it manually because the tool helped them to manage the complexity of that reuse task. This contradicts the hypothesis that the participants would give up sooner on a poor reuse task with Gilligan; whether this is a positive development or not is a matter of perspective, although it is easiest to argue that the potential payoff of a large reuse task far outweighs the minimal amount of time a reuse plan takes to formulate compared to writing the software from scratch.

> Gilligan gave me a chance with [the third task] instead of rejecting [the task] right away. This way I know for sure that the task is bad instead of going by a gut feeling. (P2)

> For the [the third task] I might have given up sooner but with Gilligan there is little risk in trying to reuse the code; it didn't matter if the task was huge because I could still keep track of what I was doing. (P4)

> [Gilligan] delays the sense of hopelessness when you're elbows-deep in someone else's code. (P10)

> I would be more willing to use the tool the more complex the reuse task; the more code I needed to reuse the more the tool would help. (P14-T1; RA-G)

G.1.3.5 Gilligan — Making Decisions: *Developers are encouraged and supported in making explicit, consistent, decisions using Gilligan.*

8 participants made 19 comments about how Gilligan encouraged them to explicitly make decisions about their reuse plan. These sentiments are related to how Gilligan presents information about

structural dependencies (Section G.1.1.5) but with the emphasis more on the decision than on the information delivery mechanism itself. By thinking in terms of decisions, and explicitly making them, the participants were able to continually encode their decisions about the reuse task within Gilligan; the resulting reuse plan was one of the main aids to the participants in maintaining and refreshing their mental model of the task itself.

I could focus on the decisions instead of being distracted by trivial details.

> Manually I would just look at the errors; with the tool I would concentrate on the actual dependencies. (P3)

> By working in the abstraction, Gilligan takes you away from the immediate compilation errors so you can focus on the decisions you are making. (P9-T2; Q-G)

> Using Gilligan is easy, I can see each decision clearly instead of being swamped by trivial details. (P15-T2; Q-G)

Using Gilligan I could focus on making the decisions, not on the mechanics that would entail.

> Gilligan was a force multiplier. Simple decisions with large consequences in the code were minor tasks due to the automation. (P1)

> [Gilligan] made it easy to focus on the decisions rather than the mechanical actions it would require manually. (P8)

Using Gilligan I could easily make aggregate decisions.

> With Gilligan I have a good representation of the task. I have a holistic view and can see what's going on; this means I can make broad, informed decisions. Here I can see right away how bad a dependency is. (P14-T3; TD-G)

> The tool makes it crazy easy to take care of large chunks of code all at once. (P15-T2; Q-G)

Gilligan encouraged me to shift my focus more towards making explicit decisions.

> Editing code is what I'm familiar with. Using gilligan I had to shift my thinking towards making explicit decisions. (P6)

G.1.4 Hypothesis Testing

All 16 participants made 76 comments that related to hypothesis testing within the context of the pragmatic reuse task. This interesting concept category revolves around the participant's desire to investigate alternative reuse decisions within their reuse plans. As none of the participants were familiar with any of the source code they were reusing they often did not make the best possible decisions on their first attempt; they made up for their unfamiliarity with the source code by trying out different options in an effort to finding the best possible option within the context of their reuse task.

One of the largest problems with investigating alternative decisions is that backing out of a decision manually required the developer to perform an overwhelming amount of work; this, coupled with their feeling that they were abandoning their initial work to get to their current state made developers feel apprehensive about changing their minds during the tasks. At the same time the participants performing the Gilligan treatments would usually investigate alternatives for any major decision; at the very least they would investigate alternative options for any expensive decision.

The participants further felt that Gilligan assisted them with investigating alternative decisions by providing the tool support necessary to quickly check different alternative decisions and immediately get feedback on the results of these decisions. This extra feedback was primarily in the form of the automated enactment of the reuse plan; the participants could quickly see the number and nature of the compilation errors that arose from the different decisions they made in their reuse plans. This extra support helped many of the developers to believe that Gilligan helped them to make the *right* decision and relieved them from some of the analysis burden of trying to make this correct decision the first time, every time. As a result of this, some participants also had more confidence in the quality of their solutions.

G.1.4.1 Reversing decisions: *Developers frequently make poor decisions while performing pragmatic reuse tasks; these are difficult to reverse manually.*

12 participants made 33 comments about their desire to change their minds about specific decisions they had made during their reuse tasks. Some decisions can be sound simple but mechanically require the developer to make a lot of changes to the code. For example, on several occasions the participant would say, "I do not care about logging" and would then proceed to remove logging references from the source code. Manually, this would involve finding every location in the code where the logger was instantiated as well as every reference to a logging field and call to the logging class; this could result in hundreds of edits, even within the context of the reuse task. Using Gilligan the participant could just reject the logger class and all of the calls and references would be automatically removed instantly for the participant.

One interesting reason participants didn't want to change their minds manually wasn't that they were worried about the work it would entail but instead related to their unwillingness to abandon all

of the investigative work they had undertook up to that point.

With Gilligan it is easy to revert any decision I have made.

> Undoing a decision was easy. It was just a couple of mouse clicks instead of commenting out lots of code or copying new code. (P1-T1; Q-G)

> We have a lot of libraries [in my company]; copying and pasting from them is very error prone. This can save time and reduce errors. It's especially faster because it's easier to backtrack. (P11)

> I was more willing to take risks [with Gilligan] knowing it would be easy to roll back. (P14)

Rolling back a decision is a lot of work to do manually.

> In the manual tasks if I made a mistake it could have large consequences; with the tool I could easily change my mind. (P2)

> If you made a misstep manually you basically have to start from scratch; with Gilligan it's easy to step back and reverse a decision. (P10-T1; RA-G)

> You could experiment a lot more [with Gilligan], if you try a class and it doesn't work out it's easier to backtrack. Manually you would have to undo lots or delete the file and start again: a lot more work! (P11)

I don't want to change my mind; I would lose all of this hard work!

> I got off on a tangent [with `QIFImport`] that cost me a lot of time. Once I was down that path I didn't want to throw all of that work away to roll back. (P10-T2; Q-M)

The level of commitment required to look at alternatives is causing to limit my exploration options.

> Manually I could not undo my decisions easily; I often wanted to investigate a different option but it was too much work. (P5)

> In the manual case there is a lot of hesitation to trying new things due to the level of commitment it would require. (P10)

Just commenting everything out is messy.

> [Manually] I would delete stuff I wasn't supposed to. In the end I switched to commenting things out, but that made a mess of the code. (P11-T2; RA-M)

G.1.4.2 Gilligan — Decision prototyping: *Gilligan encourages and assists developers in investigating alternative reuse strategies.*

11 participants made 14 comments relating to how Gilligan helped them to prototype their decisions and encouraged them to explore alternative solutions. By prototyping their decisions the participants could try several alternative reuse plans; this made it much more likely that they would make successful decisions compared to relying on their first instinct each time. In the manual treatments the participants were often unwilling to investigate alternative decisions because of the time and effort costs associated with such an investigation (Section G.1.4.1).

While prototyping decisions the participants would investigate both large-scale alternatives, such as accepting and rejecting whole classes, as well as small-scale alternatives such as accepting, rejecting, and remapping specific methods and fields. Early in the reuse task the participants would be more willing to investigate large-scale alternatives and would slowly migrate to smaller-scale experimentation as they progressed through the task.

Gilligan made it easy to prototype my decisions.

> [Using Gilligan] I can quickly see the results [of my decisions]; this is useful because it doesn't take 20 minutes to implement a single decision [as it might manually]. (P6-T1; RA-G)

> I needed to figure out how the code worked to see what I had to reuse. With the tool I could just quickly prototype my decisions and see if they worked. (P7)

> I'll just try this and see what happens. (P12-T2; RA-G)

With Gilligan I was able to make the right *decision.*

> The tool makes investigating alternatives faster than having to understand the code thoroughly enough to make the right decision the first time. It sounds bad but it's true. (P7-T2; Q-G)

> [Gilligan] breaks things down a lot more; what possibilities are there? I don't just have to copy and paste. (P8-T2; RA-G)

> Because I tried different plan alternatives [with Gilligan] I think my overall quality is better. (P14)

G.1.4.3 Gilligan — Stepwise refinement: *By helping developers progress through a pragmatic reuse task one decision at a time, Gilligan enables them to better track their successes and failures.*

12 participants made 25 comments about how they iterated on their reuse tasks with Gilligan. The primary reason for iteration was to immediately see the effects of any decision they made; in general, the participants were trying to confirm they were heading in the right decision or to catch a bad decision while it would still be easy to recover from. Iteration was also used to make the reuse task feel more manageable as the participants just dealt with one small piece of the task at a time.

Am I going the right way in this task?

> [Using Gilligan] it's easier to see the consequences of a decision. (P3)
>
> The compilation tab refreshed right away and told me if I was going in the right direction or not. (P11-T1; Q-G)

I iterated on my reuse plan to see if I needed to back out of a bad decision.

> I repeatedly iterated to see the effects of my decisions. (P4-T2; RA-G)
>
> I iterated on almost every decision I made. It gives you a decision point so you can step back and easily get out of a bad decision. (P10-T1; RA-G)
>
> It's hard to know if your decisions are bad; using Gilligan without iteration would be like working with one hand tied behind your back. (P14-T1; RA-G)

I iterated because...

> The task seemed more manageable this way. (P3-T1; RA-G)
>
> [Iterating] is much better than trying to figure it all out and having to do [the task] all at once. (P8-T1; RA-G)

G.1.4.4 Gilligan — Confidence in Solution: *Developers using Gilligan are more confident in the quality of their solutions.*

3 participants made 4 comments reflecting their feeling that they had greater confidence in their reuse task outcome using Gilligan than if they had been performing the task manually. This confidence was both applied to correctly giving up an inappropriate task as well as their solution to a successful reuse task. This confidence resulted from the participants being able to maintain an effective mental model by having a global overview of their task (Section G.1.3.2) while exploring multiple alternative solutions (Section G.1.4.2) to their reuse task.

> I felt more confident about the scope of the task [using Gilligan]; I would be more confident in predicting the eventual success or failure of a task. (P5)

> I explored the task for a shorter duration [using Gilligan] but was more confident in my decision to surrender. Manually would have taken longer but I would be less confident in my decision to give up. (P14-T3; TD-G)

G.1.5 Performing Pragmatic Reuse Tasks

All 16 participants made 91 comments that related to actually performing the pragmatic reuse plan. This concept category is somewhat less interesting than those presented in the previous four sections as it is not surprising that many of the participants comments about the experiment would center on how they actually performed the pragmatic reuse tasks. While many issues related to performing these tasks have been incorporated into the previous concept categories, four remaining themes are more specific to performing these tasks. This concept category separated into four main themes: The large amount of time still required to perform even conceptually easy pragmatic reuse tasks; the desire of participants to copy over full packages in the manual treatments; the ability of Gilligan to save participants time and effort on their pragmatic reuse tasks; and finally, on the importance of being able to easily reject source code elements not relevant to a reuse task.

The time-consuming nature of performing pragmatic reuse tasks manually is strongly linked to issues of structural dependency identification (Section G.1.1.1) and being able to quickly iterate on pragmatic reuse plans and investigate alternative reuse scenarios (Section G.1.4.1).

Participant's willingness to try to reuse whole packages at a time arose out of despair over the number of errors they were facing. This resulted from not being able to keep track of all of the details of their reuse task (Section G.1.3.1).

The time and effort savings enabled by Gilligan happened for reasons mentioned in each of the previous concept categories including giving participants access to the information they needed (Section G.1.1.5), giving them an overview of their reuse task (Section G.1.2.1), and allowing them to quickly prototype reuse plan alternatives (Section G.1.4.2).

Being able to easily remove irrelevant structural elements from the source code was very important for enabling the participants to focus on reusing source code relevant to their task rather than being overwhelmed by errors originating in code that was not relevant to their reuse task (Section G.1.3.3).

G.1.5.1 Making Work: *Most of the manual work required during a pragmatic reuse task is conceptually simple but labour intensive.*

14 participants made 20 comments about how the manual treatments were straight-forward but seemed to take much more time than they should. While the participants mentioned that both copying and pasting and locating dependencies were more time consuming than they should be, it was

observed that the participants also spent much of their time making simple low-level source code modifications.

Copying and pasting should be easy but it takes a surprising amount of time.

> Moving the types from the source to target project was more work than it should have been. (P4-T1; Q-M)
>
> [The task] was tedious due to the robotic interactions; it was very repetitive. (P8-T1; Q-M)

Just locating *the code to copy is time consuming.*

> Just finding the right code to copy and paste [made the task difficult]. (P1-T3; TD-M)
>
> [Manually] it was much harder to find and copy all of the parts of the code I wanted. (P8)

G.1.5.2 Gilligan — Saving time and effort: *Reuse tasks require less time, effort, and frustration when they are performed with Gilligan.*

14 participants made 41 comments about how Gilligan was able to save them time and effort as they completed their pragmatic reuse tasks. Several participants mentioned that automating the enactment of their reuse plan saved them a lot of the frustrating work that they encountered doing the tasks manually. Often, participants would compare the actions they were currently performing to how they would have performed the same tasks manually. These comments mainly concentrated on the ability of Gilligan to translate their understanding of the reuse task into the source code of the system. Gilligan was generally perceived as being both easier and quicker to use while investigating and performing pragmatic reuse tasks.

Automating the enactment saved me a lot of frustration.

> [Using Gilligan was] easier. I was much less frustrated. (P3)
>
> [Gilligan] saved me lots of manual work. It would have been extremely frustrating manually. (P6-T3; TD-G)

Manually this would have been a lot of work!

> Gilligan didn't force me to do anything I didn't want to do. Manually it was just more work, 2 orders of magnitude more. (P14-T2; Q-M)

[Gilligan] mostly automated the process of the reuse task. (P15)

A lot of errors that would have been a problem manually are resolved with only a click or two; this saves a lot of time and keeps things clean for you. (P16-T2; RA-G)

Gilligan helped me by significantly speeding up my reuse task.

After using the tool I wouldn't want to perform any of these tasks manually. (P3)

Without the tool I would have to prepare for this kind of task: 'Itś too late to start this today, IĺI have to get back to it tomorrow'. This is the type of task I would normally schedule a day for for one of my developers [instead of just a quick hour]. (P10-T3; TD-G)

[The task] was a good first stab; as a prototype it's a great way to go. As a first cut it would be much more effective than doing it from scratch. It's all about the classic trinity: 1) working; 2) better; 3) faster. This gets it working quickly for you and you can just go from there. (P14-T1; RA-G)

A 5 minute investment isn't that bad for a full feature! (P15-T2; Q-G)

Using Gilligan is just easier.

It was more fun to use Gilligan. (P3)

Performing the tasks were easier with Gilligan, even though I used it for a harder task. (P8)

The manual approach may be easier when going through your own code but with 3rd party code it is multiple orders of magnitude easier. (P9)

G.1.5.3 Large Copies: ***Developers are likely to copy entire packages while manually performing a reuse task, even though they know it is likely to backfire on them.***

7 participants made 12 comments about their temptations to copy large batches of source code while performing their reuse tasks manually. This need generally arose from the huge number of errors the participants were facing; they hypothesized that by copying large amounts of code they could resolve many problems in one quick operation. While most participants also vocalized that it was a bad idea due to the amount of extraneous code they would be reusing, many participants still followed this course of action to try to resolve their problems.

I'm just going to copy this whole package.

> I should just be copying full packages at this point. (P2-T2; RA-M)
>
> The dependencies are spread out throughout half of the source project. After a while you get deep into the `engine` and `engine.util` [packages]; you think you should take the whole package to save time but you can't tell if all of the classes are necessary. (P16-T1; Q-M)

I'm really tempted to just copy it all.

> I should have gone the other way: copied everything over and just started deleting like mad! (P2-T2; RA-M)
>
> I just can't find stuff. I'm tempted to copy everything but that probably won't help and I'd probably get a lot of extraneous stuff. (P13-T2; RA-M)

G.1.5.4 Gilligan — Rejecting Elements: *Being able to easily remove elements that are not related to the reuse task is essential to reusing only relevant source code.*

8 participants made 18 comments about removing unrelated source code elements during reuse tasks. These elements frequently occur in pragmatic reuse tasks as the code that is being reused is often not modularized in a way that only provides the functionality the developer wants to reuse. In the Related Artist task in particular, the participants had to remove many methods and fields (10 of 13 from `AudioScrobblerService`) that were not relevant to the functionality they were tasked with reusing.

Many of the manual participants struggled with balancing the work of removing irrelevant elements against the effort of resolving the compilation errors for the same elements; the participants using Gilligan always just removed those elements they did not need as this was a straight-forward operation using the tool. Only one participant felt that Gilligan should have more fine-grained support for rejecting elements; in this case the participant wanted to reject a dependency from one class but not another (as decisions in Gilligan are consistent across the system, this request is not possible).

I'm able to get rid of code using Gilligan that I would keep around manually because it would be too much effort to get rid of it.

> Instead of doing a lot of onerous manual deletion of this functionality I don't want, I'll just copy the class to make the errors go away. (P1-T3; TD-M)
>
> Using Gilligan was like the difference between using a scalpel and an axe [to get rid of unwanted code]. (P14)

> Often the code is interspersed with lots of extraneous stuff you don't want; removing this cruft could take all day so you just keep it. (P14)

> If you're not paying attention, the manual approach can be a lot more dangerous than with the tool. With the tool I can more easily trim things down. (P15)

It's really easy to get rid of structural elements I don't need.

> The ability to reject elements at a fine-grain level was more effective than doing it manually. (P4)

> With Gilligan it's easy to get rid of extraneous stuff. (P10-T1; RA-G)

More fine-grained accept / reject support is required.

> More fine-grained approach is required for this task. (P4-T2; RA-G)

G.2 Prompted Concept Categories

The three prompted concept categories are comprised of 16 themes and 66 sub-themes. The first two originated in questions that were asked of the participants, one before the study and one after. Before the participants started they were asked some background questions about pragmatic reuse tasks. After they had finished they were asked how they thought Gilligan could be further improved in the future. Finally, a large number of task-specific observations were made as the participants worked through the experimental treatments.

The results for these three categories are reported more quantitatively than in the last section; the participants' answers were grouped and enumerated and are reported in more succinct form than for the organic concept categories.

G.2.1 Pragmatic Reuse: Rationale, Impediments, and Frequency

During the initial introduction to the experiment the concept of the pragmatic reuse task was described to each participant. This description was given to ensure that each participant was familiar with the terminology that was used for the rest of the experiment. Participants were then asked three main questions: "Do you perform pragmatic reuse tasks? If so, how often?", "Why do you perform pragmatic reuse tasks?", and "What impediments are there to performing pragmatic reuse tasks?". The participants were free to answer these questions in any way they chose. These three questions provide some general background about the participant's thoughts about pragmatic reuse before the experiment began.

G.2.1.1 "Do you perform pragmatic reuse tasks? If so, how often?"

Each participant provided one answer for this question. The answers they were able to give was not bounded, these three categories were distilled from their responses (summarized in Table G.2). The categories are vague as the participants found it difficult to estimate how often they performed pragmatic reuse tasks in practice. While some of the participants recalled specific pragmatic reuse tasks, none of them could give a meaningful quantitative estimate as to how often they performed these tasks.

Pragmatic Reuse Frequency	# of Participants
Frequently	6
Sometimes	7
Rarely	3

Table G.2: Frequency that participants perform pragmatic reuse tasks.

G.2.1.2 "Why do you perform pragmatic reuse tasks?"

Participants were able to give as many answers as they wanted for this question; their answers were not given with a specific rank or order. This was an open ended question; the resulting 5 categories were delineated during the card sort. If a participant gave multiple answers that fell into a single category, they were only counted once. Their reasons are summarized in Table G.3.

Most of these reasons are self-explanatory, but I will expand upon two of them.

One frequently occurring reason for performing pragmatic reuse tasks involved reusing code as an exemplar for their current task. The participants used these exemplars for various reasons. One reason was that they would be working from example code that wasn't fully functional but demonstrated how to use interesting APIs. Another reason for reusing code in this manner was to simply use the code as a learning resource rather than something they would end up executing. The final reason involved situations where they had code that provided functionality similar to what they needed, but wasn't exactly right. In these cases they would reuse the code as an exemplar and change it to meet their needs.

Another reason for performing pragmatic reuse tasks cited by the participants involved preserving existing encapsulation within their system. In these cases the participants would reuse source code from within their own system to either preserve the existing encapsulation mechanisms within their system or to enable them to make minor modifications to the code without worrying about breaking the parts of the system that relied on the initial code.

Pragmatic Reuse Rationale	# of Participants
The code I need already exists.	9
It is faster than writing the code from scratch.	9
To use the existing code as an exemplar.	9
It is easier than writing the code from scratch.	6
To preserve existing encapsulation of the existing code.	3

Table G.3: Participant's rationale for performing pragmatic reuse tasks.

G.2.1.3 "What impediments are there to performing pragmatic reuse tasks?"

Participants were able to give as many answers as they wanted for this question; their answers were not given with a specific rank or order. This was an open ended question; the resulting 5 categories were delineated during the card sort. If a participant gave multiple answers that fell into a single category, they were only counted once. Their reasons are summarized in Table G.4.

The main impediment to pragmatic reuse tasks perceived by the participants regarded whether or not the amount of work required to perform the task would outweigh the expected benefits of the reuse task. This danger originated from the fact that without actually performing the task the participants did not know of any other way to estimate the difficulty of a reuse task. While propagating the changes was also a concern, some participants expressed that they would not actually want changes propagated to them as their usage of the code might invalidate the need for any future changes to the originating source code.

Pragmatic Reuse Impediments	# of Participants
Uncertainty: Will it take a lot of work to reuse the code?	6
Propagating future changes can be difficult.	4
Keeping variable names consistent between systems.	3
Risk reusing badly-written code.	2
Might reuse source code you don't understand.	2

Table G.4: Impediments to pragmatic reuse tasks identified by participants.

G.2.2 Gilligan: Suggestions for Improvement

The participants were also queried about, "How could Gilligan be improved in the future?" and "What UI problems did you have while using Gilligan?". The feedback from these questions gave interesting insight that will be used to further improve the tool in the future. 15 participants suggested ways to improve Gilligan and UI issues that interfered with their usage of the tool. Two other themes

also emerged here: First, some participants found that they had to be careful not to be lured into an "autopilot" mode using the tool; while this could help them get the code compiling they still needed to have an understanding of the task in order to successfully complete the reuse task. Secondly, the participants made several comments about Gilligan's learning curve. The participants all felt that it wasn't overwhelming and by the end of the experiment they were mainly comfortable and effective at using the tool.

G.2.2.1 Gilligan: Requests for Enhancement

At the conclusion of the experiment the participants made several suggestions as to how Gilligan could be improved. These have been distilled into 6 individual suggestions (these are summarized in Table G.5).

For of these suggestions would be straightforward to implement and would improve Gilligan's ability to help developers perform pragmatic reuse tasks. On larger tasks, specifically the 3rd task in the experiment, the performance of the structural views was "sluggish". This could easily be improved with better caching and lightweight prediction of the dependency counts and the transitive closures in the system. Several participants wished there was an explicit way for them to track their progress in the reuse task; these participants wanted Gilligan to record the size of the validation view and the number of compilation errors after each run of the extractor. Optimally these numbers could be graphed so the participants could visually see their progress over time and easily revisit decisions that had a significant impact on their progress. Some participants also wanted validation and checkout to be combined into a single action; this would be trivial to implement and from the observations during the experiment would be a good idea. Some participants wanted the editor to be better linked with the structural views; they wanted the structural view contents to update automatically based on the code they selected in the editor. This is a good idea and should be added to the system.

The request for a dependency recommender was a good suggestion but would be difficult to implement; this is a suggestion that will be examined in future iterations of the tool.

Two participants requested incremental build support be added to Gilligan. This means they would like the ability to modify the reuse code without having to worry about their changes being overwritten the next time the reuse plan was enacted. This would be a difficult change to implement and I do not believe it is a good idea. The current version of the tool forces the developers to encode all of their decisions within the reuse plan; Gilligan further ensures that these decisions are consistently applied across the system. By enabling the developer to work with the code directly the reuse plan can be more easily bypassed, making the reuse plan harder to understand and enabling the developer to begin to make inconsistent decisions within their reuse task.

Suggested Gilligan Improvement	# of Participants
Increase navigation performance.	3
Provide an explicit indication of progress.	3
Link validation and enactment.	3
Link the editor with the structural views.	3
Support incremental building.	2
Add a dependency cost recommender.	1

Table G.5: Suggested improvements for Gilligan.

G.2.2.2 Gilligan: UI Issues

After the experiment the participants made several comments about specific user interface choices that they did not like while performing their pragmatic reuse tasks (these are summarized in Table G.6). With the exception of "the views can be confusing", each of these issues can be easily improved in the system. While the participants sometimes found the views confusing, they were generally effective at using them at the conclusion of one or two reuse tasks.

The validation view was designed in "backwards" fashion to enable developers to see information about how a dependency is used (the structural views only show what a dependency *uses*, not how it *is used*). The participants found this behaviour inconsistent with the structural views and can easily be changed for future versions of the tool.

The text of labels that the developer has not yet investigated in the structural views were displayed in a light-grey text to help them determine those dependencies they had investigated compared to those that still existed for them to investigate; however, this meant that "interesting" dependencies that still needed to be viewed were less obvious than those that had already been investigated. A new way of showing this piece of information should be developed for future versions of the tool.

Gilligan User Interface Shortcomings	# of Participants
Validation view is backwards.	5
Selection behaviour is awkward.	4
Views can be confusing.	4
Greyed-out text can hide details.	2

Table G.6: Gilligan user interface shortcomings.

G.2.2.3 Gilligan — Trust: *Trusting the tool can be dangerous.*

Two participants made one comment each about how trusting the tool too much can be dangerous:

> I had a stronger tendency to go into an autopilot mode when I was using Gilligan. (P15)

> The dependency counts are discouraging; if you're looking at a high count you're more likely not to import something. This might be good for compilation but not for getting the reuse code to work. (P16-T2; RA-G)

Participant 6 in particular had trouble trusting the tool. This participant made 6 comments at various points during the experiment about their difficulty in "trusting" and "relying" on the Gilligan. While the participant successfully completed reuse tasks both manually and with the tool, they were clearly uneasy with the tool support that tried to automate so much of the development process for them.

G.2.2.4 Gilligan — Learning Curve: *Over the course of the experiment I was able to get comfortable using Gilligan for pragmatic reuse tasks.*

6 participants made 10 about Gilligan's learning curve. All 6 participants remarked that while Gilligan definitely had a learning curve, they felt comfortable using it by the end of the experiment. This was a matter of concern at the outset of the experiment: was Gilligan easy enough to use that developers could succeed at their reuse tasks using it? While the participants were given a short training task at the beginning, they mainly figured out how the tool worked as they worked through their experimental tasks, even with this approach all the participants felt reasonably comfortable with Gilligan by the conclusion of the experiment.

> Once I was used to the tool it was fine; getting used to it wasn't unreasonable at all. (P3)

> I'm used to Eclipse and [Gilligan] didn't seem that much different. (P13-T1; Q-G)

> The task only took 5 minutes, even with still learning the tool. (P15)

G.2.3 Observations

During the card sort, 9 themes and 40 sub-themes were identified. This section reports only the 5 most interesting themes and one catchall for general observations under which the remaining themes were amalgamated. The remaining themes focus on several interesting observations that were made during the experiment.

The first examines the disparity between the frustrations experienced between the participants undertaking the two 'good' reuse tasks using either Gilligan or doing them manually. The second theme examines how the different participants felt about the suitability of each task they were asked to perform. The third theme discusses the role of the scale of the dependencies in a system to influence a reuse task. One interesting theme was that participants performing the Torrent Downloader task using Gilligan often had premonitions that the task was poor early in their investigation, although they chose to pursue it long after they had made these observations. The final theme examines the role the test harness had on the participants while they were performing their tasks.

G.2.3.1 Manual Problems With Good Tasks

The QIF Parser task and the Related Artists task were both examples of good reuse tasks. 4 participants expressed frustration with the Related Artists task while performing the manual treatment. In general, their complaints stemmed from the fact that there was so much code that needed to be removed from the system that was unrelated to the functionality they were trying to reuse. These comments are significant as they represent tasks where the participant wanted to give up during the manual treatment, even though they were working on what was ultimately a good reuse task; this situation did not arise for any participant performing a task using Gilligan.

> There are lots of errors and I don't know where they're all coming from. This task was tedious and frustrating for me. (P13-T2; RA-M)

One participant also had significant difficulties with the QIF Parser task; this task was well modularized and was generally a good task, but the participant ended up down a bad path that was difficult to get out of.

> [I have] endless cycles of dependencies and errors. When there's this many I can't really focus on identifying the main problem with the code I have reused. (P16-T1; Q-M)

G.2.3.2 Task Evaluations

After completing each task the participants were asked, "Was this a good reuse task?". For clarification they were provided with, "If you needed to provide the same functionality yourself, would you reuse that code?". Table G.7 enumerates the answers to this question. Both the QIF Parser task and the Related Artists task were included in the study as 'good' reuse tasks. The majority of participants, 14 and 12 respectively, agreed with this assessment. The Torrent Downloader task was included as a 'bad' task; 13 of the participants concurred with this assessment.

Between the two 'good' reuse tasks, 7 of 32 participants performing those tasks (22%) believed the tasks to be poor candidates for reuse. 6 of the 7 participants made this decision while performing a manual treatment.

Participants who failed at a reuse task were likely to report that the reuse task was a poor choice; for the two 'good' reuse tasks, 5 of the 7 participants who concluded the reuse task was a bade idea also failed to complete the task.

Participant 6 reported that the Related Artists task was a poor choice after successfully completing the task with Gilligan; their rationale for this decision was that, 'You could just examine the code and write it yourself'. Participant 16, while performing the task manually, reported that the QIF Parser was a poor task because, 'The dependencies are spread out throughout half of the source project'. The participant believed the code to be poorly structured because they followed an erroneous path for

much of the task and ended up reusing much more code (7785 LOC compared to an average of 98) LOC for this task) than the other participants needed to reuse for the same functionality.

All 16 participants commented on how difficult the Torrent Downloader task was, and on how poor the code seemed; however, three participants still reported that they thought it was a good reuse task. Even though they failed to get the task to work, all three thought that trying to reuse the Azureus code would still be faster than writing a compatible BitTorrent engine from scratch.

	Good Task	Bad Task
QIF Parser		
Manual	6	2
Gilligan	8	0
Total	*14*	*2*
Related Artists		
Manual	4	4
Gilligan	7	1
Total	*11*	*5*
Torrent Downloader		
Manual	2	6
Gilligan	1	7
Total	*3*	*13*

Table G.7: # of participants who felt the task was good or bad.

G.2.3.3 Scale of Dependencies

Table G.8 lists the number of participants who commented that the scale of the structural dependencies was problematic for them as they pursued their reuse task. The maximum number for any cell in the Table is 8. None of the participants using Gilligan for the first two tasks felt overwhelmed by the number of dependencies they needed to consider during their reuse task. For the third task 5 participants performing the Gilligan treatment identified the scale of the dependencies as the main reason for their failure; the manual participants were more likely to blame other reasons including the extreme number of editors and compilation errors they were forced to consider.

	# of Participants
QIF Parser	
Manual	1
Gilligan	0
Total	*1*
Related Artists	
Manual	3
Gilligan	0
Total	*3*
Torrent Downloader	
Manual	3
Gilligan	5
Total	*8*

Table G.8: # of participants who felt the scale of dependencies was problematic.

G.2.3.4 Torrent Downloader (Task 3): Early Signs of Failure

The Torrent Downloader task was a "poison pill". It was inserted as a bad task to see if Gilligan could help a participant infer that a task was destined to be a failure. While participants using Gilligan did not give up statistically significant less amount of time, 7 of the 8 participants using Gilligan for this task made comments early in the reuse task where they predicted the task would ultimately be a failure.

> *After 7 minutes:* The tool is probably give me clues I shouldn't be doing this task, but I can see where I'm going. I don't think this is a good reuse task. (P2-T3; TD-G)

> *After 12 minutes:* I expect this is setting me up to include the entire system. (P10-T3; TD-G)

> *After 15 minutes:* There are too many dependencies; this results in an explosion of the transitive closure. It's like running up scree. (P14-T3; TD-G)

5 participants (4 using Gilligan and one performing the task manually) made comments about how the tool helped them to decide to give up on the third reuse task.

> I suspect I would have more success with Gilligan; It would have given me the information I needed faster. (P7-T3; TD-M)

> [Automating the enactment] didn't give me the warm fuzzy 'you're doing better' feeling I got during the earlier tasks; it just wasn't getting any better after 10–15 iterations. But maybe that was telling me something too. (P10-T3; TD-G)

G.2.3.5 Test Harness Influence

Providing participants with the test harness potentially interfered with their perception of the reuse tasks. 5 participants made comments about how they used the test harness to guide their investigative efforts manually while one participant did the same using Gilligan. The ability to access the test harness was given to participants for both treatments to try to neutralize any benefits this may have provided. While the participants were asked not to rely on the test harness at the outset of the experiment, they were not restricted from doing so during their tasks. The manual participants were more generally used the test harness as a way to ground themselves whenever they needed to step back to try to get an overview of their reuse task; as Gilligan provided this automatically (see Section G.1.3.2) the tool-supported participants did not need to use the test harness this way.

> The test harness gave me a good starting point. (P5-T1; RA-M)

> I was implicitly guided by the functional test harness; in the manual case this really drove me and influenced how I did the task. (P14-T2; Q-M)

G.2.3.6 General Observations

In addition to the observations mentioned above an additional 160 observational notes were made about all 16 participants as they proceeded through their tasks. These observations could be broken down into four themes (and further into 21 sub-themes): Gilligan-specific observations (24), manual-specific observations (27), task-specific observations (18), and general observations (91). While there is a wealth of data in these notes, trying to tease anything useful from them proved to be a fruitless exercise. As such, none of the data is reported here although it is used throughout the evaluation as necessary to make specific points.

Appendix H

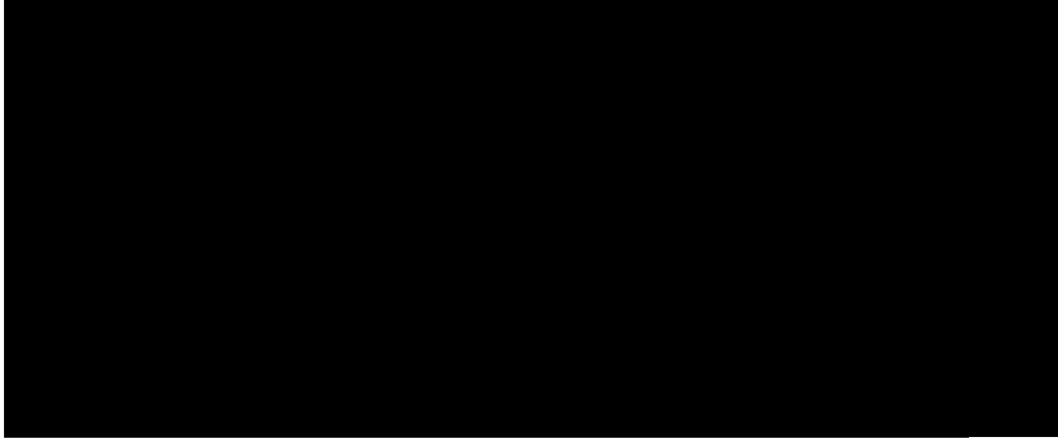
Calgary Research Ethics Board Approval

CREB 5005: *Pragmatic software reuse* (p. 238).
CREB 5605: *Pragmatic software reuse* (p. 239).

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical Conduct in Research Using Human Subjects"*. This form and accompanying letter constitute the Certification of Institutional Ethics Review.

File no:	**5005**
Applicant(s):	**Reid T. Holmes** Robert J. Walker
Department:	**Computer Science**
Project Title:	**Pragmatic Software Reuse**
Sponsor (if applicable):	



Distribution: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary

CERTIFICATION OF INSTITUTIONAL ETHICS REVIEW

This is to certify that the Conjoint Faculties Research Ethics Board at the University of Calgary has examined the following research proposal and found the proposed research involving human subjects to be in accordance with University of Calgary Guidelines and the Tri-Council Policy Statement on *"Ethical Conduct in Research Using Human Subjects"*. This form and accompanying letter constitute the Certification of Institutional Ethics Review.

File no:	**5605**
Applicant(s):	**Reid T. Holmes**
Department:	**Computer Science**
Project Title:	**Pragmatic Software Reuse**
Sponsor (if applicable):	

Distribution: (1) Applicant, (2) Supervisor (if applicable), (3) Chair, Department/Faculty Research Ethics Committee, (4) Sponsor, (5) Conjoint Faculties Research Ethics Board (6) Research Services.

2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 • www.ucalgary.ca