Systematizing Pragmatic Software Reuse

REID HOLMES, University of Waterloo
ROBERT J. WALKER, University of Calgary

Many software reuse tasks involve reusing source code that was not designed in a manner conducive to those tasks, requiring that ad hoc modifications be applied. Such *pragmatic* reuse tasks are a reality in *disciplined* industrial practice; they arise for a variety of organizational and technical reasons. To investigate a pragmatic reuse task, a developer must navigate through, and reason about, source code dependencies in order to identify program elements that are relevant to the task and to decide how those elements should be reused. The developer must then convert his mental model of the task into a set of actions that he can perform. These steps are poorly supported by modern development tools and practices.

We provide a model for the process involved in performing a pragmatic reuse task, including the need to capture (mentally or otherwise) the developer's decisions about how each program element should be treated: this is a *pragmatic-reuse plan*. We provide partial support for this model via a tool suite, called Gilligan; other parts of the model are supported via standard IDE tools. Using a pragmatic-reuse plan, Gilligan can semiautomatically transform the selected source code from its originating system and integrate it into the developer's system.

We have evaluated Gilligan through a series of case studies and experiments (each involving industrial developers) using a variety of source systems and tasks; we report in particular on a previously unpublished, formal experiment. The results show that pragmatic-reuse plans are a robust metaphor for capturing pragmatic reuse intent and that, relative to standard IDE tools, Gilligan can (1) significantly decrease the time that developers require to perform pragmatic reuse tasks, (2) increase the likelihood that developers will successfully complete pragmatic reuse tasks, (3) decrease the time required by developers to identify infeasible reuse tasks, and (4) improve developers' sense of their ability to manage the risk in such tasks.

Categories and Subject Descriptors: D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*; D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments*; K.6.3 [**Management of Computing and Information Systems**]: Software Management; H.1.2 [**Models and Principles**]: User/Machine Systems—*Software psychology*

General Terms: Human Factors, Experimentation, Algorithms, Reliability, Languages

Additional Key Words and Phrases: Software reuse, white box reuse, pragmatic software reuse, source code investigation, lightweight process and tooling, low commitment, planning, pragmatic-reuse plans, pragmatic-reuse plan enactment, Gilligan

ACM Reference Format:
Holmes, R. and Walker, R. J. 2012. Systematizing pragmatic software reuse. ACM Trans. Softw. Eng. Methodol. 21, 4, Article 20 (November 2012), 44 pages.
DOI = 10.1145/2377656.2377657 http://doi.acm.org/10.1145/2377656.2377657

Professor David Rosenblum handled the reviewing process.
This work was supported in part by a Discovery Grant and a Collaborative Research and Development Grant from the Natural Sciences and Engineering Research Council of Canada, and by a Faculty Award from IBM. This work was performed at the University of Calgary.
Authors' addresses: R. Holmes, David R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, Ontario N2L 3G1, Canada; email: rtholmes@cs.uwaterloo.ca; R. J. Walker, Department of Computer Science, University of Calgary, 2500 University Dr. NW, Calgary, Alberta T2N 1N4, Canada; email: walker@ucalgary.ca.

DOI 10.1145/2377656.2377657 http://doi.acm.org/10.1145/2377656.2377657

1. INTRODUCTION

Software reuse approaches encourage the development of software systems using preexisting artifacts instead of creating them anew. 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].

The majority of software reuse research has focused on preplanned reuse approaches, such as object-oriented inheritance [Dahl and Nygaard 1966; Snyder 1986; Johnson and Foote 1988], software components [McIlroy 1968; Szyperski 2002; Ravichandran and Rothenberger 2003], and program families or software product lines [Parnas 1976; Krueger 2000]. Preplanned reuse approaches suffer from three main drawbacks:

(1) the notorious difficulty in predicting which pieces of software should be built in a reusable fashion [Tracz 1990; Gaffney and Cruickshank 1992; van Gurp and Bosch 2002];
(2) the economic infeasibility in developing all software in a reusable fashion [Gaffney and Cruickshank 1992; Biggerstaff 1994; 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; Lions et al. 1996].

In contrast to the reuse scenarios with which preplanned reuse approaches are associated, developers often find themselves in a position where a development task they are performing is familiar to them: either they have previously implemented the functionality that their task requires, or they have access to existing source code that provides the functionality they need. 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 preplanned reuse techniques [Garlan et al. 1995]. In these situations, the developer has three main options:

(1) to reimplement the functionality;
(2) to refactor the original system to make the functionality directly reusable; or
(3) to reuse the functionality's source code in an ad hoc, copy-and-modify manner.

There are numerous drawbacks to each of these options. Reimplementing the functionality is expensive and does not leverage any testing or other positive attributes associated with mature source code. Refactoring the original source code [Griswold and Notkin 1993; Opdyke 1992] is often not feasible for a number of organizational and technical reasons [Cordy 2003; Kapser and Godfrey 2008]: 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 risk the introduction of defects by refactoring something that is known to work already; or 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

Reusing source code that was not designed explicitly for reuse has been known by many monikers: code scavenging [Krueger 1992], ad hoc reuse [Prieto-Díaz 1993], opportunistic reuse [Rosson and Carroll 1996], and copy-and-paste reuse (also known as cut-and-paste or copy-and-modify) [Lange and Moher 1989]. We choose to introduce the term *pragmatic reuse* because each of these previous labels has negative connotations,

while pragmatic reuse tasks can be appropriate and effective [Rosson and Carroll 1996; Cordy 2003; Kapser and Godfrey 2008].

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." Frakes and Kang [2005] identify two impediments to pragmatic reuse tasks: (1) development tools may not be effective at promoting pragmatic reuse; (2) such reuse efforts are hampered by the lack of a structured process. Other researchers have identified similar shortcomings [Krueger 1992; Sen 1997; Morisio et al. 2002; Ravichandran and Rothenberger 2003].

Pragmatic reuse versus preplanned reuse need not be an either/or proposition; for example, a component has to be initially created, and to do so by pragmatically reusing high-quality functionality could garner the best of both worlds. We also note that both preplanned and pragmatic reuse approaches possess common drawbacks: assumptions about their context may not be explicit or, at least, not automatically checkable. For example, the 1996 explosion of the Ariane 5 rocket resulted from the black-box reuse of functionality in a context where the functionality's assumptions did not hold (specifically, that the velocity of a rocket could be contained in a 16-bit memory location) [Lions et al. 1996]; while this assumption could have been made explicit, it is unrealistic to expect that all assumptions are explicit—as a trivial example, who would think to record that there are seven days in the week? In addition, both approaches necessitate locating the functionality for reuse: preplanned approaches assume that a component exists that either fits perfectly or that the target system can be adapted to make it fit; a pragmatic approach assumes that the functionality itself is a legitimate target for modification.

We believe that, by systematizing pragmatic reuse tasks, developers are more likely to try—and succeed—at them, increasing the amount of source code that is reused in industrial organizations.

1.2. About This Article

In this article we describe an approach for planning and performing pragmatic reuse tasks that mitigates previously identified impediments. This approach, embodied in a tool suite called Gilligan, enables developers to systematically investigate the boundaries of the source code that they want to reuse. As they investigate the code, decisions that they make—about which elements to reuse and how to reuse them—are captured in the form of a *pragmatic-reuse plan*. It is important to note that this decision problem balances delineating the useful functionality (i.e., a feature location problem) with the cost of truncating dependencies at different points. The developer may decide to trim away some "useful" functionality or to take a bit of extraneous functionality, if he perceives this as simplifying his overall task.

While building an understanding of the task is crucial to its success, a pragmatic-reuse plan in isolation would capture only the developer's intent for the task, leaving him to manually extract and integrate the code they want to reuse—a highly tedious and error-prone process. To reduce this burden, we also semi-automate the extraction and integration phases of pragmatic reuse tasks by leveraging the knowledge captured in the pragmatic-reuse plan. This support has the added benefit of simplifying *hypothesis testing*: by automating the (approximate) enactment of the plan, developers can quickly change their reuse plan to test alternative decisions; this allows them to experiment to find the best reuse plan for their task, making exploration a low-risk activity. Such hypothesis testing is otherwise not feasible to perform, due to the effort required and high likelihood of introducing errors during the manual equivalent.

We have iteratively developed our approach over a few years, evaluating each iteration quantitatively and qualitatively using a series of questionnaires, case studies, and controlled experiments. Industrial developers were involved in the development of the tool and in our evaluations.

It is important to note that our approach is not intended to supplant but to complement others. In particular, we do not aid the developer in initially locating source code for reuse, nor do we provide much support for making the individual decisions that our pragmatic-reuse plans will record—this is functionality that their IDE already provides or that could be provided by program comprehension tools—we merely make it clear which decisions must be made, which decisions have been made, and what those decisions were. While our approach helps the developer to initially select and integrate some existing functionality, it does not: (a) enforce conventions or constraints in the target system—standard refactoring tools should be applied at this step if desired; (b) necessarily enable the developer to reuse poorly designed, "spaghetti" code—but it will let him see sooner that this is the situation in which he finds himself; nor (c) automatically support the reuse of associated test suites from the original system—this remains as future work.

Our approach aims to reduce some of the risks inherent in pragmatic reuse tasks in the absence of specific support, thereby making them more attractive when appropriate. However, this is not an attempt to provide the final word in tool support. We outline several directions for improvement and future study.

Our previous work has introduced the model of pragmatic-reuse plans along with an initial tool to support it [Holmes and Walker 2007a]; demonstrated that developers have difficulty exhaustively identifying dependencies [Holmes and Walker 2007b]; and shown that support for semi-automatically enacting pragmatic-reuse plans improves their performance [Holmes and Walker 2008]. In this article, we draw together these earlier results into a comprehensive and coherent narrative, while detailing the significantly different design that our tooling utilizes as a result of industrial feedback. We report on a previously unpublished, formal experiment, finding that our approach (1) significantly increases the chances of a developer successfully completing a pragmatic reuse task, (2) significantly decreases the amount of time these tasks take, (3) significantly increases developers' ability to discern whether a particular task is feasible, and (4) improves developers' sense of their ability to manage the risk in such tasks.

The remainder of this article is organized as follows. Section 2 gives a motivating example of a pragmatic reuse task. Our model for pragmatic reuse tasks is described in Section 3. Related work is discussed in Section 4, to argue that none suffices for our problem. Section 5 describes the Gilligan tool suite for planning and enacting pragmatic reuse tasks, which reifies our model. We overview the historical evolution and evaluation of Gilligan in Section 6. A comprehensive evaluation of our prototype's ability to support the pragmatic-reuse planning and enactment process is described in Section 7. Section 8 discusses remaining issues.

2. MOTIVATIONAL SCENARIO

Consider a developer working on a Global Positioning System (GPS) visualization system called UltiGPX. UltiGPX provides a simple visualization of the latitude and longitude of points on a hikers' route (Figure 1(a)). The developer of UltiGPX decides to provide a profile view of the route representing the hiker's changes in elevation. While planning this feature, she recalls a similar visualization within another system: Figure 1(b) shows a screenshot of the Azureus BitTorrent client,[1] a peer-to-peer file transfer client that happens to have a network throughput visualization. The developer

[1]http://azureus.sf.net, v2.4.0.2.

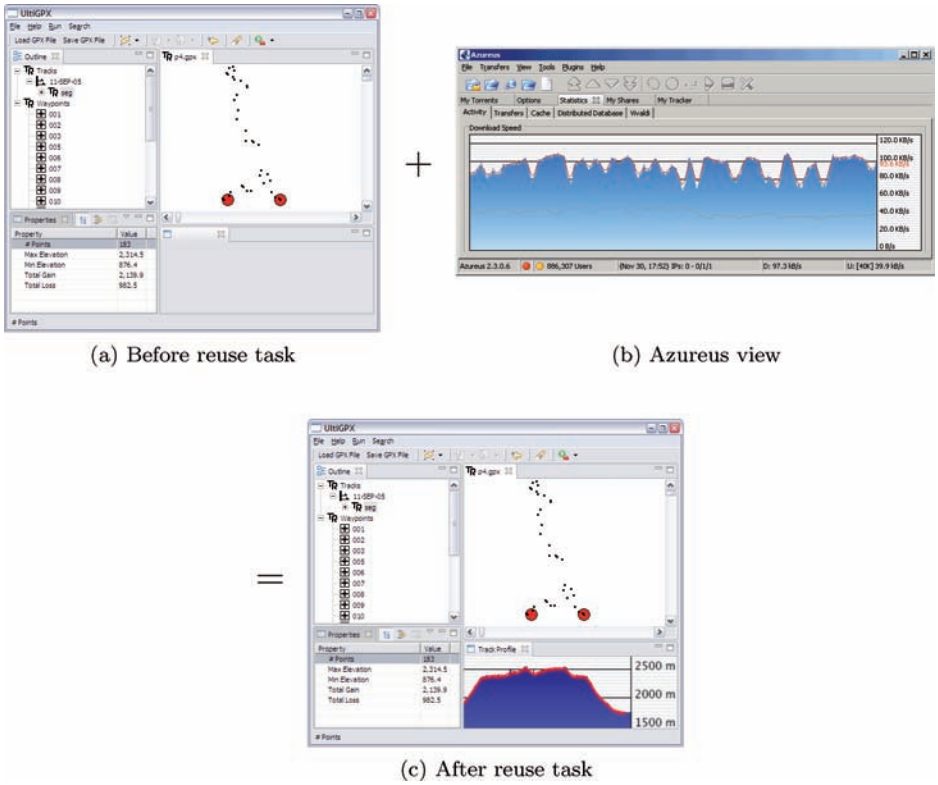

Fig. 1. Extending the UltiGPX application through pragmatic reuse.

decides that the Azureus view approximates the functionality desired and decides to investigate its reuse. Figure 1(c) shows a screenshot of UltiGPX with the profile feature added by reusing the Azureus code.

2.1. Exploration

Since Azureus was designed to support peer-to-peer file downloading, not to provide reusable application programming interfaces (APIs) for its user interface widgets, it is not immediately obvious that the reuse task will be tractable. The developer investigates the source code manually within an integrated development environment (IDE). First, she uses the IDE search features to locate a part of Azureus involved in network visualization; this search quickly leads her to the `graphics` package,[2] in which `SpeedGraphic` seems likely to be the most relevant class.

The developer begins her exploration within the `drawChart(..)` method of `SpeedGraphic`, as it sounds promising. 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 referenced, and methods called. She 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,

[2]The fully qualified package is org.gudy.azureus2.ui.swt.components.graphics.

she determines that 7 of the types are common to both UltiGPX and Azureus (these both use the SWT framework) meaning that these need not be considered further. But the developer must look more critically at the 7 remaining types to determine their relevance to the task.

`SpeedGraphic.drawChart(..)` makes one call to both the `enter()` method and the `exit()` method from the `AEMonitor` class. The `AEMonitor` class seems simple enough, but *if* the developer looks at the class that it extends (`AEMonSem`), she finds more than 700 lines of threading code, which reference many other classes and that do not seem relevant to her task. Through further investigation, the developer determines that reusing `AEMonitor` would likely involve reusing all the dependent classes in its transitive closure, the scope of which would overwhelm the utility of reusing the `SpeedGraphic` code in the first place. Instead of reusing the `AEMonitor` class, she decides to 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, she notices many references to constants in the `Colors` class. These references roughly duplicate functionality she already has in UltiGPX: it defines its own colour constants. Instead of reusing the 28 fields in the `Colors` class, she remaps the 13 references to the 9 colour constants referenced by the code to be reused to UltiGPX's own colour constant fields.

2.2. Analysis of the Task

The `SpeedGraphic` functionality has been widely tested and deployed and can be considered mature: it had undergone 40 revisions over four years prior to this task. The code had also been widely deployed as Azureus had been downloaded more than 200 million times. While the task ended up reusing only 373 lines of code (LOC) of the 223 kLOC that comprise Azureus, the fact that the code was of high quality made the developer feel more comfortable than she would have with her own first attempt at creating such functionality.

While the structure of the reused code is relatively simple, the developer had to investigate more than 20 different source files to make decisions about the importance of each of them. Tracking these decisions while investigating unfamiliar source code can be daunting. To actually perform the task she had to copy those relevant classes from Azureus to UltiGPX, modify the copied files as necessary to remove rejected elements plus calls and references to these rejected elements, and update any remapped elements to elements within UltiGPX. While carrying out the task is conceptually straightforward, it is difficult for the developer to remember all the decisions made while navigating between these various source files. Indeed, when she started to do the task, she had to revisit several files to remember what decisions she had made. The developer never actually knew if she had completed her investigation: she may have missed an important dependency when she was navigating the various files and may not have found out about it until she actually attempted the reuse task.

This task would have been easy to dismiss out of hand due to the domain differences between Azureus and UltiGPX, the manual effort needed on the developer's part, and the high risk since it was unclear that the task would be successful until completed. Nevertheless, it was ultimately an effective way to reuse a high quality source code feature. Dismissing it out of hand would have been an opportunity lost.

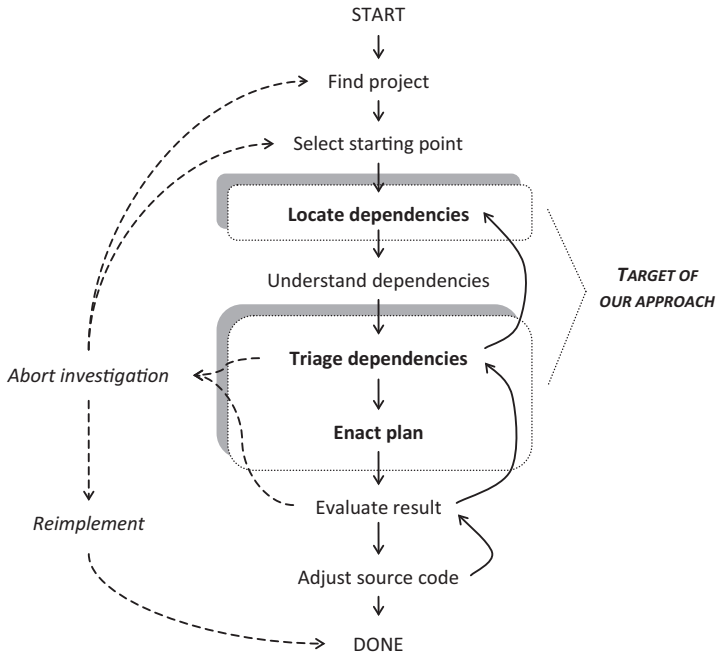

Fig. 2. Our model of the pragmatic reuse process. The tool support that we describe in Section 5 focuses on the portion of the model indicated by the rounded rectangles.

3. A PRAGMATIC REUSE PROCESS MODEL

From the motivational scenario, a more general model for the investigation and planning of a pragmatic reuse task emerges, as follows (see also Figure 2).

(1) The developer initially locates a project deemed to contain potentially useful functionality, to serve as the originating system for reuse.
(2) The developer locates an element within this project, which serves as the starting point for investigation, that embodies the desired functionality.
(3) By traversing dependencies between program elements, the developer seeks to eliminate elements that constitute functionality that is either unneeded, already present, or more easily reimplemented in their system. Each of these "triaging" decisions is recorded (in a *pragmatic-reuse plan*)—mentally, on paper, or electronically—in order that they can be revisited and revised as the developer acquires more knowledge.[3]
(4) The pragmatic-reuse plan is enacted (manually or via tool support) by performing the inferred modifications.
(5) The developer determines whether the results are satisfactory, by examining the integrated source code, any compilation errors, or the results of any tests.
(6) If the results are satisfactory, the developer may still need to make adjustments to the reused source code.

[3]These decisions must be made on the basis of the developer's understanding of the originating system and the target system. Whether this understanding is better acquired through an in-depth, global, reverse engineering process or a local, lightweight skimming of the code associated with each dependency (or some combination thereof) is an important decision in itself that depends on a cost/benefit analysis of the particular task. Factors influencing such a decision include the complexity of the systems, whether the target system is safety-critical, whether the pragmatic reuse task is being undertaken as a first step, etc.

(7) If the results are unsatisfactory, the triaging decisions can be revised, and the enactment redone.
(8) At any point, the process can be abandoned to either select a different starting point, a different originating system, or to reimplement the functionality from scratch.

There are two basic strategies that we have observed developers use in manually undertaking pragmatic reuse tasks: *analyze-then-act*, where the source of interest is considered within the original system and only those parts deemed most useful are copied to the target system and then modified to deal with dangling references; *cut-and-stanch-the-bleeding*, where the developer copies source deemed relevant to the task, and attempts to repair the dangling references (which appear in the Eclipse IDE as red underlining in the source code) by either modifying that source or by copying yet more.

3.1. Challenges

A manual approach to pragmatic reuse brings with it a number of challenges for the developer, corresponding to the various steps in our model and the transitions therein. Several of the challenges may be viewed, at first glance, as either trivial or insurmountable, but neither is the case.

Manually identifying dependencies is problematic due to the fact that these are not made explicit in IDEs. Both strategies mentioned above suffer from similar issues: the developer must visually identify dependencies from a background of irrelevant details in the source code. Working from an abstract view of the dependencies (as in a graph) eliminates its context, making details of their use and extent invisible.

Having identified a dependency, the developer must (1) remember its location, (2) consider what is depended upon, (3) perform potentially nonlocalized reasoning about the rationale behind the dependency, (4) consider the relevance to the target system of this rationale, and (5) determine the impact to the task and target system in either accepting or rejecting the depended-upon element.

In addition to the potential burden of these five steps for a single dependency, the fact that they must be repeated for each dependency leads to the need to also remember all these points, with the possible need to recall them for revision and definitely for enactment. Developers sometimes attempt to use pen and paper to record their decisions, but we have witnessed this low-fidelity support being used inconsistently and, worse, it tends to be slow and easy to make mistakes therewith.

Manual enactment is further complicated by the need to make a large number of small edits throughout the codebase that are consistent with the decisions having been made. Revision after manual enactment is difficult as undoing the manual edits is not an atomic operation. Performing undos on individual files can lead to backing up too far.

We can categorize the challenges in the manual process as follows.

CH1: *High memory burden*. A large number of decisions have to be remembered for later enactment and possible revision. Worse, since such revision requires also the recall of rationales for earlier decisions, details of the codebase discovered during analysis have to be remembered as well.
CH2: *Information overload*. The desire to both apply key knowledge to all pertinent locations, while also being thorough and complete within a particular source element, are competing demands for the developer's attention. He must both focus on triaging the immediate dependency but also on whether the overall task is going well. And he must ignore the large amount of information that is irrelevant to his task.

CH3: *Difficulty in understanding a dependency and its context.* Dependencies can have complex interactions and the original rationale for their presence can be hard to reconstruct, particularly when one's attention is divided.

CH4: *Difficulty in estimating the cost of the task a priori*. A developer generally must guess up front whether the potential pragmatic reuse task will be of low enough cost to justify the degree of benefit to be gained therefrom. Thus, he will generally either avoid a task out of fear, despite the fact that the task could work out well, or begin performing the task, meaning he is less willing to abandon his investment in time and effort even when it becomes apparent that significant problems are arising.

CH5: *Inability to check for completeness*. The developer can never be sure that she has analyzed all the dependencies in the reused elements. Compilation errors that result from copying the source code may be due to this incompleteness or to mistakes made in the manual enactment process. A lack of such problems is no guarantee that the developer has made a conscious decision about a given dependency.

CH6: *High cost of editing and undoing*. Frequent, scattered, small changes lead to developer fatigue. Errors propagate as a result. Backing up and starting again can be painful, and further mistakes can accrue at this point.

CH7: *Difficulty in expressing intent*. Because standard IDE tools are not tailored to pragmatic reuse tasks, the developer must constantly translate her intent into the generic actions supported by those tools, like browsing through the source code, or using a hyperlink to travel to a depended-upon declaration.

We now proceed to examine the related literature for potential solutions and evidence concerning our model.

4. RELATED WORK

The argument that any source code duplication activities are negative has long been a staple of folklore (e.g., see Hunt and Thomas [1999]). Nevertheless, research into industrial practice provides evidence that it can be both necessary and positive, as explored in Section 4.1. There are a variety of cognitive issues inherent in the pragmatic reuse process that should be taken into account; these are examined in Section 4.2. There are also various approaches that might be used to support different parts of the pragmatic reuse process; we discuss these in Sections 4.3–4.5, but none suffices for our needs.

4.1. Pragmatic Reuse is Common, Necessary, and Positive

Lange and Moher [1989] performed an in situ observational study into the activities of a professional software engineer developing software in an object-oriented environment. They found that his development was predicated on a pragmatic reuse strategy: 85% of the new classes developed during the one-week of observation involved copying-and-pasting of existing code, followed by heavy modification.

Rosson and Carroll [1996] report on a qualitative study of expert Smalltalk programmers reusing user interface classes in small graphical applications. They identified the "reuse of uses" (a form of pragmatic reuse) as an effective and frequent strategy that was employed.

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—despite the organization's heavy investment in preplanned reuse.

Through a small survey on industrial developers (12 industrial developers from 6 organizations), we earlier reached four key findings [Holmes and Walker 2007a]: (1) they

perform pragmatic reuse tasks; (2) they have access to large amounts of source code; (3) they reuse code to save time and improve quality; and (4) they want to understand a feature's dependencies before they reuse them. That survey, in part, drove this research.

Ncube et al. [2008] have revived the term *opportunistic software systems development* (OSSD) to give it a positive connotation, in a special issue of *IEEE Software* on the subject. OSSD is generally described as an ad hoc customization process, in contrast with the composition of commercial-off-the-shelf components. In this special issue, Jansen et al. [2008] describe the important role of "pragmatic and opportunistic reuse" in two start-up companies; they claim that it was necessary in order to rapidly develop innovative software products, which otherwise would not have been built. None of the work in this special issue helps to support the model we have described.

Brandt et al. [2009] report on both an in-laboratory observational study of developer habits and a post hoc analysis of developers' queries to a web portal; among other findings, they report that pragmatic reuse is a common practice.

Of course, the mere fact that time and again industry is seen performing pragmatic reuse is, by itself, no indication that the practice is a warranted or disciplined one; many in academia have witnessed frequent, ill-advised copy-and-paste activities conducted by undergraduate students, so perhaps it should be a practice that is stamped out. Countering such concerns, Cordy [2003] provides many reasons why industrial organizations reuse code via clones instead of refactoring, and Kapser and Godfrey [2008] conducted studies that demonstrate that pragmatic reuse tasks are commonplace and beneficial. Three primary rationales are raised: refactoring does not immediately better an organization's financial situation; the risk associated with refactoring a system may not be acceptable; and the redundancy provided by clones isolates subsystems better.

Even within a single system, Kim et al. [2005] have found that clones are frequently short-lived, and when they are long-lived they are not easily refactored. 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 such a process is an active research topic [Kim and Notkin 2006; Duala-Ekoko and Robillard 2010].

4.2. Cognitive Aspects of Reuse

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

Norman [1998] presents a model of the psychology of an individual involved in conducting a task (the "seven stages of action") that can be compared to our task model and the challenges we have identified. The seven stages of action comprise (1) the formation of goals; three stages involving *execution*: (2) the formation of intention to act, (3) the specification of a sequence of actions, (4) the execution of that action sequence; and three stages involving *evaluation*: (5) the perception of the state of the world, (6) the interpretation of that perception, and (7) the evaluation of those interpretations in terms of the goals. The model is cyclical until the goals are satisfied. Norman points to potential challenges in this process when applied to concrete contexts: the *Gulf of Execution*, which is the difficulty involved in operationalizing one's intentions through a set of allowable actions; and the *Gulf of Evaluation*, which is the difficulty involved in interpreting one's perceptions and evaluating those interpretations in terms of the goals. Our seven challenges can be seen as concrete manifestations of Norman's Gulfs: high memory burden (CH1), information overload (CH2), high cost of editing and undoing (CH6), and difficulty in expressing intent (CH7)

all relate to the Gulf of Execution; difficulty in understanding a dependency and its context (CH3), and difficulty in estimating the cost of the task a priori (CH4) relate to the Gulf of Evaluation; and inability to check for completeness (CH5) relates to both.

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 work focused on the issue of copying code within a single system; instead, our primary intent is to enable developers to reuse larger-scale functionality from external systems more effectively.

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. By helping developers to 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 the low-level details that performing the task might involve.

A number of general approaches have been proposed to aid developers to avoid getting lost during program investigation tasks, such as recording the paths that they have traversed [Janzen and De Volder 2003] and automatically filtering the list of programmatic elements presented to the developer [Kersten and Murphy 2005]. No such approaches suffice for pragmatic reuse, but we use the lessons learned from them to inform our design.

The cognitive challenges with which developers must contend while performing pragmatic reuse tasks are not unique; they are common to many software maintenance and evolution tasks. A comprehensive overview of these challenges is given by Storey et al. [2000]: as developers seek to gain an understanding of the system they are investigating, they can reason about the system in three main ways. In a top-down investigation, the developer starts with high-level concepts that they understand, and map these into the source code [Brooks 1983]. When performing a bottom-up investigation, developers start with the source code and iteratively form higher-level abstractions about the design of the system [Shneiderman 1980]. Hybrid approaches involve both top-down and bottom-up investigations wherein the developer frequently switches between high-level concepts and the low-level source code [von Mayrhauser and Vans 1995].

Developers planning and performing pragmatic reuse tasks employ techniques related to each of these reasoning approaches. When first investigating a system looking for a feature to reuse, developers sometimes employ a top-down approach to identify the part of the system containing the functionality they desire; however, as noted by Soloway and Ehrlich [1984], this approach is less effective when the code being investigated is unfamiliar—often the case in pragmatic reuse tasks.

Conversely, developers can work in a bottom-up mode, building program and situation models to understand their tasks [Pennington 1987]. These models enable developers to reason about their tasks, and how they will perform them, more abstractly. Unfortunately, the scale of software systems greatly impedes these bottom-up approaches as the developer can be overwhelmed by the detail present in the source code, inducing them to miss important dependencies [Soloway et al. 1988; Briand et al. 1999; Vanciu and Rajlich 2010].

Hybrid approaches that enable developers to work in both a top-down (to deal with scale) and bottom up (to elucidate salient detail) have also been proposed. In particular, Littman et al. [1986] note that developers use either systematic or as-needed approaches while investigating systems. This is an effective strategy for performing pragmatic reuse tasks; developers focus on high-level structures as much as possible, descending into the source code only when necessary. Von Mayrhauser and Vans [1995] further emphasize that developers frequently switch between different levels of abstraction; tool support to ease and facilitate these transitions would reduce the

burden on developers performing pragmatic reuse tasks. Finally, Letovsky [1986] postulates that developers perform inquiry episodes, whereby they pose a question and perform in-depth analyses to confirm or refute their hypotheses; Sillito et al. [2008] have empirically observed such behavioral patterns. Pragmatic reuse tasks frequently take this form as developers seek to confirm that specific dependencies are relevant to the functionality the developer wants to reuse.

Pragmatic reuse tasks may be inhibited by developers' unfamiliarity with the original source code and its domain. Soloway and Ehrlich [1984] show that (procedural) programs written in a poorly structured manner (i.e., without conformance to underlying plans) are difficult to understand because they fail to support the developer's knowledge about the role of such structure; these sorts of programs appear least likely to lead to successful pragmatic reuse tasks. For software evolution tasks, Littman et al. [1986] and Koenemann and Robertson [1991] found that developers attempt to comprehend source code only as much as needed to accomplish their immediate purposes: when that purpose is satisfied with a cursory examination, they seek no further; our observations of developers during pragmatic reuse tasks echo this. Some studies of software reuse [Burkhardt and Détienne 1995; Rouet et al. 1995] show that, when functionality is located, information about the situation from whence the functionality comes is sought or inferred, including design rationales, constraints, and structure. Burkhardt et al. [2002] demonstrate that even novices are able to effectively construct a mental model (the "situation model") of source code to support a reuse task, despite unfamiliarity with the functionality's situation.

To successfully enable pragmatic reuse tasks, developers should be supported in the construction, maintenance, and evaluation of the model of the task they are investigating. This support should enable developers to gain a cohesive high-level overview of their task while not restricting them from investigating alternative levels of abstraction when they deem it appropriate. The goal of this support is to relieve some of the burden of navigating large, unfamiliar systems, thereby enabling developers to focus on their tasks without becoming lost or overwhelmed.

4.3. Potential Support for the Location Phase

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 system permits a developer to locate an example that provides the functionality from which he could then copy some source code into his own system. Strathcona only locates examples, so while it might be useful in the early phases of the model shown in Figure 2, it provides no support for the core activities.

A number of more recent approaches (e.g., Bruch et al. [2009]) attempt to recommend specific edits to code based on mining of software repositories. These approaches operate at a much smaller scale, and are not appropriate for locating significant pieces of functionality.

Other systems have looked at creating reusable components from existing code. Several approaches do not consider the target system [Caldiera and Basili 1991; Lanubile and Visaggio 1997; Inoue et al. 2005] and so, these are inappropriate to the requirements. CodeGenie [Lemos et al. 2009] extracts code slices from existing systems based on test cases selected by the developer; while this is an interesting alternative approach to pragmatic reuse, it would not immediately help with our motivational scenario.

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]; Robillard [2008]). 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.

S6 [Reiss 2009] is a tool that automatically locates a set of source code fragments, selects one of them, and integrates it into a target system; it leans heavily on structural cues and the developer's ability to write exact test cases. S6 is necessarily too restrictive to use for arbitrary pragmatic reuse tasks: it operates at a within-class granularity, rather than automatically extracting entire features; its dependence on test cases to drive the process would inhibit its ability to apply in situations where the developer only possesses a notional sense of the functionality he desires (as in our motivational scenario).

4.4. Potential Support for the Investigation and Planning Phases

Rigi [Müller and Klashinsky 1988] and Ciao [Chen et al. 1995] both statically analyze the source code of a system and allow a developer to navigate the structural dependencies contained therein. Navigation of structural dependencies is not sufficient for pragmatic reuse tasks. Further, for larger tasks, a means for *reducing* the information at hand is needed [Fischer 1987; Robillard et al. 2010].

Some details of our pragmatic reuse model are similar to those given by Robillard and Murphy [2007] for their concern graph model. While concern graphs and pragmatic-reuse plans contain the same structural elements, with the same relationships between them, their intent diverges significantly. Concern graphs aim to bring together a set of structural elements to aid program understanding; 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.

Robillard [2008] has investigated providing tool support that can recommend relevant program elements to the developer based on the elements she has previously investigated. We have used recently pursued preliminary work in combining this technique with our pragmatic reuse model [Holmes et al. 2009], but that is strictly an extension of the work presented here. Recent work by Lawrance et al. [2008] have shown the potential value in modeling developer navigation behaviour via the theory of information foraging; this could provide an alternative recommendation approach for us.

4.5. Support for the Enactment Phase

Transformation-based approaches to reuse were prevalent in the 1980s, for example, that of Feather [1989]. Similarly, approaches like that of Gouda and Herman [1991] and of Yellin and Strom [1997] automatically adapt components to new contexts. However, such approaches require complete, formal specifications to operate that are not applicable to the lightweight process inherent in pragmatic reuse tasks.

Low-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 that arises from extracting fragments from the originating system, but these would still require semantic- and conceptual-level support for the developer while investigating and planning their task.

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 perform pragmatic reuse, dependencies sometimes need to be eliminated or replaced.

While many approaches have advocated refactoring [Griswold and Notkin 1993; Opdyke 1992] code into reusable APIs, this is not always possible. The original code may no longer be maintained or its maintainers may not be willing to refactor their code

to meet the new requirements [Cordy 2003; Kapser and Godfrey 2008], particularly for reuse by an external project.

Cottrell et al. [2008] have developed the Jigsaw tool that allows one method to be copied and integrated into another existing method. Their approach effectively integrates methods, identifying the commonalities and differences between a source and target method. 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 that of our motivational scenario as it only integrates individual methods. It might be useful during the enactment phase of the model, but would not help with the investigation and planning phase.

5. THE GILLIGAN TOOL SUITE

To overcome the impediments to pragmatic reuse that have been previously identified, we have created the Gilligan tool suite. Section 5.1 describes the high-level design goals for Gilligan and how they relate to the challenges identified in the pragmatic reuse process (Section 3.1). Section 5.2 describes how pragmatic-reuse plans are reified in Gilligan. Section 5.3 describes how the tool suite supports identifying the reuse boundary in unfamiliar code. Section 5.4 outlines the actions that the developer can take within Gilligan. Section 5.5 describes how Gilligan manipulates the reused code to semi-automatically integrate it into the target system.

5.1. Design Goals

When designing Gilligan we had several design goals, as described below, with respect to planning (PL) and enacting (EN) pragmatic reuse tasks. These design goals, and the resulting tool, have been specifically tailored to industrial developers performing pragmatic reuse tasks; they have changed in response to feedback from industrial developers over time (see Section 6). Each of these goals is intended to address one or more of the challenges described in Section 3.1 (noted with CH).

- PL1: *Provide an abstract representation of the structural elements and relationships being investigated.* Source code contains a wealth of information; unfortunately, the majority of this data is not relevant when creating a pragmatic-reuse plan. Gilligan provides an abstract view to help developers focus on planning their reuse task, rather than overwhelming them with detail. (CH1, CH2)
- PL2: *Promote easy navigation between structural elements.* Developers do not navigate arbitrarily between files of a system when delineating the reuse boundary. Developers typically follow outgoing dependencies and are often interested specifically in an element's direct dependencies and their transitive closure. (CH3, CH7)
- PL3: *Explicitly record decisions about structural elements.* Manually tracking the plethora of decisions made when planning a reuse task is difficult; supporting the explicit recording of these decisions ensures consistency, helps provide a high-level overview of the reuse task, and enables automation of the enactment of the plan. (CH1, CH2, CH4, CH5, CH6)
- PL4: *Encourage systematic performance of pragmatic reuse tasks.* When navigating unfamiliar code, it is easy to revisit the same program elements repeatedly or to miss program elements altogether. Gilligan guides developers to relevant program elements they have not yet visited, and visually emphasizes nodes they have already triaged. (CH1, CH2, CH3, CH5, CH7)
- EN1: *Automatically perform as much extraction and integration as possible.* Performing a pragmatic reuse task involves a huge number of simple program transformations (find an element, copy it, determine where it should go, paste it, modify it trivially, repeat). Automating this can both save time and reduce errors. (CH6, CH7)

EN2: *Support an iterative workflow.* Developers cannot be expected to make every decision before enacting a reuse plan. Gilligan supports an iterative workflow that encourages investigating alternative reuse decisions and seeing their concrete effects in the reused code. (CH1, CH2, CH3, CH5, CH6)

5.2. Pragmatic-Reuse Plans

Our model's chief artifact is the *pragmatic-reuse plan*, which we use to refer interchangeably to the conceptual plan and our reification thereof. We reify pragmatic-reuse plans in a graph structure: each of the structural elements (classes, methods, and fields) in the originating system is represented as a node in the graph. These nodes are linked by edges corresponding to their static relationships. A pragmatic-reuse plan maintains a list of structural elements that have optional pieces of metadata associated with them. Each structural element can have multiple relationships to other elements and can be tagged with at most one reuse intent decision. Every structural element is identified by its fully qualified name; classes are additionally attributed with an identifier for their originating system. Currently, the relationships between elements in the reuse plan consist only of static structural relationships; data dependencies are not considered. Furthermore, we have only investigated pragmatic reuse tasks involving source code written in the Java programming language to-date, and pragmatic-reuse plans are tailored accordingly.

Pragmatic-reuse plans encode five kinds of relationships between structural elements.

—*Inherits.* Inheritance relationships are recorded between pairs of types. Classes can extend at most one other class and implement multiple interfaces. Interfaces can only implement other interfaces. Inherits relationships are represented as a directed edge from the subtype to the supertype.
—*Calls.* Method call relationships are recorded whenever a method is referenced within a program. Method calls include calls to type constructors, static methods, and super methods. Method calls generally involve 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. Directed edges point from callers to callees.
—*References.* Field references can occur both within methods and within static initializers. Like calls relationships, references relationships are represented by directed edges from the referee to the referent.
—*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. While method return types and parameters have types as well, these dependencies are captured through calls made to the parameters or return types rather than using has-type relationships.
—*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., to find all the calls made by all the methods in class `Foo`).

5.2.1. Decisions a Developer Can Make about a Structural Element. Our model enables the developer to encode one of six triage decisions for a structural element. 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 that 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 structure of the reused code and its structure in its originating system.
—*Reject.* Structural elements that represent functionality the developer does not want to reuse can be tagged as rejected. If a class is rejected, all its contained methods and fields are also rejected; however, individual methods and fields can be rejected from 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 to which he would like the element to remap. 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 specified by the developer.
—*Already Provided.* Common structural elements between the originating source system and the target system 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 pragmatically reused to satisfy the dependencies of any accepted code as they already exist in the 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. The ability to make this triage decision was added based on feedback from industrial developers (see Section 6).
—*Inject.* Another special case added at the request of industrial developers was the ability to inject any arbitrary fragment of code into a class being reused. They wanted to do this so that they could add their own stubs to the source code to which remapped fields and methods could then point. This enabled them to effectively reject an element while keeping references to it live in the reused code.

Each of the developer's decisions affects how much the reused source code must be altered when the pragmatic reuse task is enacted. Any edge from an accepted element to a rejected element will result in a dangling dependency when the pragmatic reuse task is enacted; 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 that he has accepted. Dangling dependencies arising from rejected elements represent source code statements that the developer does not consider core to the functionality he wishes to reuse and that will be removed when he enacts the reuse task.

5.3. User Interface

The Gilligan perspective within the Eclipse IDE comprises a series of views arranged to help a developer to plan and to perform a pragmatic reuse task on a single screen without switching between different programs; a screenshot of the default interface[4]

[4]It is important to note that, being implemented as a plugin to the Eclipse IDE, the developer is free to move views around, close views, or open additional ones.

Fig. 3. Gilligan default UI; A–C are the planning views, D is the annotated source code, E is the validation view, F is the standard error view, and G is the tool pallet.

is provided in Figure 3. The UI has four main components: the navigation views (A–C), the annotated source code view (D), the validation view (E), and the error view (F). We describe each in turn.

5.3.1. Navigation Views. The navigation views are the primary means for the developer to explore the source code he is investigating for reuse to determine the relevance of each structural element to the functionality he wishes to reuse, and to assess the cost associated with reusing it. Each of panels A–C comprises four columns: a collapsible tree-based list of dependencies, a decision column, and counts of the direct and of the indirect dependencies for each element. We decided to use tree-lists of dependencies to support PL1, enabling the developer to focus on the dependencies rather than details in the source code.

Elements are nested in the tree by containment; the maximum nesting depth of this list is three (package, class, method or field), as every package is indicated by its fully qualified name at the top level. (Inner types are handled by prepending the containing type's name to that of the inner type at the second level. Anonymous types are treated as part of the implementation of the containing method body, and cannot be directly manipulated; their dependencies are treated as dependencies of their containing method. Generics are handled via their type erasure.) The icons for each element in the tree are consistent with the rest of the IDE for familiarity. We also use icon overlays to provide additional information for class elements; these icons show if a class has super- or subclasses, and if the class has hidden elements. Elements in the tree are shown with a light grey colour if they have not been investigated; once the developer clicks on them, their text becomes black, providing a simple visual cue about whether an element has been investigated or not (PL4).

Developers' decisions about program elements are explicitly tracked; a decision can be recorded for an element by selecting it and opening the IDE's context menu, choosing the desired decision. These decisions are described in more detail in Section 5.2.1. The four main decisions (accept, reject, remap, and already provided) each have a

distinctive color (green, red, blue, and yellow). These visual cues inhibit developers from unnecessarily triaging the same element repeatedly, enabling them to quickly scroll through the lists to visually gain a high-level perspective of their reuse task (PL3).

The direct and indirect columns were added to help developers gain a sense of the potential difficulty involved to reuse an element: an element with a single direct dependency and no transitive dependencies will likely be easier to reuse than an element that has four direct dependencies and one hundred transitive dependencies. The numbers also direct developers to those dependencies that will have the largest impact on the ultimate success of their reuse plan (PL4).

The leftmost panel (A) displays a list of all the elements that the developer has made decisions about while planning her task; an element is automatically added to this panel when a decision is recorded for it. Arbitrary elements can also be added to this panel, which permits the developer to identify one (or more) elements at the outset to investigate for reuse before making any decisions.

Once an element is selected in the leftmost panel (A), the center panel (B) is populated with its direct dependencies (PL2). If the selected element is a field then it is dependent only on its type; if a method is selected then it is dependent on any method it calls and field it references; if a class or interface is selected then the direct dependencies are the union of the direct dependencies of all its methods and fields, as well as its superclasses and implemented interfaces. Being able to quickly identify the direct dependencies of an element is crucial to assessing how coupled it is to its originating system; while one might expect developers to be able to do this effectively using just an editor, we have found this not to be the case [Holmes and Walker 2007b]. Dependencies that exist in libraries are also shown in the navigation views, although any dependencies the library elements may have are not shown.

If any element is selected in the center panel (B), the rightmost panel (C) is populated with the element's transitive closure of dependencies (if more than one element is selected, the union of the transitive closures is given). This view ultimately gives the developer guidance as to the effort required to reuse an element (the more dependencies an element has, the harder it is to reuse), as well as the applicability of an element (if most of the elements seem unrelated to the desired functionality, perhaps the selected element is not relevant either). Without Gilligan, developers frequently try to generate this information manually, but collecting the data involves switching between many editors and keeping track of the details is difficult; in contrast, this view can be populated and manipulated immediately (PL1 & PL2).

Developers can revise their triage decisions at any time. If a developer decides to reject an element that they previously accepted, the element will no longer be included in any enactment or validation. Triage decisions are not cascaded, however; that is, if an element is rejected and some of the elements that depended on that element were accepted, these will remain accepted. While this could mean that extraneous elements are accepted, these elements may also be reachable by other elements that are themselves explicitly accepted. The decision not to cascade elements was chosen to ensure that developers were aware of each triage decision in their plan; having Gilligan instead infer their intention could introduce sources of potential uncertainty about how a triage decision was made.

5.3.2. Source Code View. While the navigation views are the main means for navigating through the system, developers frequently wish to view the source code (D) to gain additional context for making their decisions. The text in the editor is annotated with the decisions that the developer has made so far; this enables the developer to examine the text while still remaining within the context of their reuse plan (PL3 & PL4).

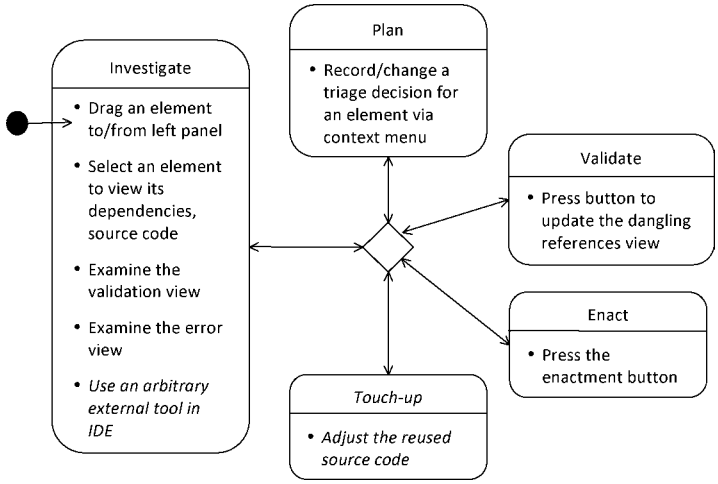

Fig. 4. An informal activity diagram outlining the actions that the developer can perform during a pragmatic reuse task. There are very few constraints on the developer's order of actions. As Gilligan is integrated with the Eclipse IDE, arbitrary tools can be used at any time; in particular, touch-up (the final, manual stage of enactment) necessitates using other parts of the IDE.

5.3.3. Validation View. The validation view (E) contains a list of elements[5] that the developer still needs to triage in order to proceed with his task (PL4). As developers annotate elements with accept decisions, they must also decide how to triage the direct dependencies of those elements. Whenever the developer selects the green checkmark button (in G), the validation view is updated. The right half of the validation view displays which elements require the selected element. This is the only place that Gilligan displays incoming dependencies; this was specifically requested by our industrial participants to make the rationale for an element being in the view more evident.

5.3.4. Error View. The standard Eclipse error view (F) is also displayed; this list is restricted to errors in the developer's project, not the source project for the reused code. As the developer enacts the reuse plan (by selecting the shopping cart icon in G), this view updates with the errors that remain in the developer's project. The ordering of the elements in this list is specified by Eclipse.

5.4. Supported Workflow

A developer using Gilligan to perform a pragmatic reuse task has five basic activities at her disposal (see Figure 4):

(1) *investigate*, to understand the nature of the dependencies from accepted elements to others;
(2) *plan*, to record or change a triaging decision about a dependency within the pragmatic-reuse plan;
(3) *validate*, to determine whether the plan contains dangling references;
(4) *enact*, to cause the modifications implied by the plan to be carried out; and
(5) *touch-up*, to adjust the reused and integrated source code to deal with minor issues that the tooling cannot.

[5]This list is currently ordered alphabetically, which echoes the ordering of Eclipse's standard Problems view. While one might imagine that some other order would be preferable (like how many elements depend on the element), ultimately the developer ought to triage *all* these elements before proceeding to enactment.

There are almost no constraints on the order of these activities; the developer can switch between them at will, encouraging an iterative and exploratory workflow. The only constraints are trivial ones: dependencies cannot be investigated until a starting point is identified and the element of interest is selected; decisions cannot be planned until a starting point is identified; a plan cannot be enacted until it contains at least one decision; a plan cannot be validated until it contains at least one decision.

Investigation within Gilligan operates by: the selection of program elements in order to view their direct and indirect dependencies, and their annotated source code; the examination of the validation view; and the examination of the error view that is populated by the compiler after enactment. Investigation can also be conducted externally to Gilligan, with the IDE's other tools (although no participant in any of our studies ever did so). The touch-up step must be conducted externally to Gilligan.

By iterating on their plans, developers can investigate alternative reuse decisions in a way that would be prohibitively expensive otherwise. By enabling developers to explore options, Gilligan can give the developer greater confidence in the effectiveness of their plan, as well as encouraging them to reuse smaller amounts of code (as it is easy to reject an element to see the results before accepting it) (EN2).

5.5. Enacting Pragmatic-Reuse Plans

While planning a pragmatic reuse task without tool support can present the developer with an overwhelming amount of information to consider, enacting a pragmatic reuse task without tool support instead forces the developer to perform an overwhelming number of essentially trivial source code modifications [Holmes and Walker 2008]. Manually performing these tasks involves a simple cycle. First, the developer must find the element that should be reused in the originating system and copy it. Second, he must locate where the code should be pasted in the target system. Third, he must manipulate the reused code to resolve dangling dependencies and other simple compilation problems. This cycle must be repeated for every element that the developer has decided to reuse when creating his reuse plan. While these steps are simple, they are numerous (hundreds of them in even small reuse plans), and provide ample opportunity for a developer to introduce additional errors into his system.

Gilligan tries to bridge the gap between the intent of the reuse plan and the realization of the task by semiautomatically enacting the pragmatic-reuse plan for the developer (EN1). First, Gilligan migrates source code fragments from the originating project into the target project according to the pragmatic-reuse plan (Section 5.5.1). Second, Gilligan mitigates the compilation problems in the reused code that arise from removing it from its originating environment (Section 5.5.2).

Developers can enact their reuse plan at any time. If the validation view is empty, there will likely be few compilation errors; however, even when the validation view is populated and dangling dependencies result from enactment, these errors can help developers to choose the order in which they approach the elements in the validation view. The enactment process is an atomic and fast action. This atomicity makes enactment very lightweight, encouraging developers to quickly iterate on their decisions and enact the plan, rather than planning for when they are willing to make an investment in enacting their plan. As such, incremental enactment is not currently supported, nor do we foresee a need for it.

5.5.1. Extraction. When the developer initiates the enactment of the pragmatic-reuse plan, Gilligan 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 pragmatic-reuse plan. It then migrates the source code associated with these nodes into the target project. 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 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 source code in the developer's project and makes it easy to rollback from the reuse task by just deleting the generated folder (EN2). When code is extracted, the structure of its original package hierarchy is maintained for ease of comprehension. The extraction process is complete, that is, partial rollback is not supported. This means that any changes made to the reused code will be overwritten when the plan is next enacted. This was done to encourage developers to modify their reuse plan (wherein decisions are all consistent), rather than make piecemeal changes to the reused artifacts before they considered their plan complete.

We slightly restrict which nodes must be copied from one system to another. Any accepted type will be migrated along with any untriaged types that contain accepted methods or fields. After Gilligan has copied the code into the target project, dependencies between the reused classes will remain valid as the package structure was maintained. Any dependencies to structural entities outside those being reused would normally cause compilation problems; however, the integration portion of Gilligan resolves many of these.

5.5.2. Integration. During the integration phase, the source code that has been migrated from the original system to the target system must be manipulated to resolve 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 internal representation of the pragmatic-reuse plan, Gilligan can precompute each of the changes that the tool should perform to repair dangling references, as the plan implicitly contains a static picture of every entity within the system that the reused code references. From this, Gilligan predetermines which relationships are being deliberately severed or remapped in the reuse plan, and based on the annotations in the plan, it determines how to manage these dangling references (EN1).

Gilligan proceeds in four major steps. First, it handles new code being added to the reused entities (Section 5.5.2.1). Second, it updates dangling references (Section 5.5.2.3). Third, it removes any unnecessary code (Section 5.5.2.2). And last, a simple finalization step takes place (Section 5.5.3).

5.5.2.1. Managing source code additions. The code addition step adds new code to those entities previously migrated to the target project (Section 5.5.1). 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). The burden is on the developer to ensure that the injected fragments are syntactically and semantically correct in the context of a class body (e.g., injecting fields or methods is usually fine, but injecting an arbitrary fragment will likely result in an error).

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. References to these fields are updated in the next step.

5.5.2.2. 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.) 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. Gilligan rejects field references and method calls only at the statement level.[6] Gilligan can manage dangling dependencies to libraries (e.g., jar files), although accepted library elements are not copied during the extraction phase.

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 that they are interested in from an inheritance hierarchy. Any number of supertypes can be removed. If the subclass were dependent on a method within a rejected supertype, this would arise 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 five cases: calls to injected and remapped methods and references to injected, extracted, and remapped fields.

5.5.2.3. 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. The import statements are also updated in this step to reflect changes in dependencies due to removals.

5.5.3. Finalizing Source Code Modifications. Each of the changes made by the three previous steps are made to the AST representation of the code,[7] not directly to its text. After all the steps are complete, Gilligan writes the changes to the files, collecting statistics about the scope of the changes it has made. This final step also deals with minor issues such as preserving the formatting of the source code.

6. GILLIGAN: EARLY PROTOTYPES AND EVALUATIONS

Gilligan was developed and evaluated using three main prototypes, in series; the third prototype is as described in Section 5. Here, we outline the design of the earlier two prototypes, the key findings from their evaluations, and how this impacted the design

[6]Eclipse restricts where comments can be inserted into abstract syntax trees (ASTs).

[7]By means of the support for AST manipulation provided by the Eclipse JDT.

goals for our approach and our final prototype. The first author conducted all the studies and analyzed all the results.

6.1. First Prototype

Initially, our design goals included two points that we have since abandoned: (PL5) visualize all the structural elements and their relationships; and (PL6) provide a high-level overview of the reuse task. We also did not consider enactment (EN1 & EN2) to be a significant concern originally, but only focused on planning. As a result, our first prototype involved the visualization of the graph representing the pragmatic-reuse plan and provided no support for enacting the reuse plan.

Four industrial developers, each from different companies, applied the first Gilligan prototype to a pragmatic reuse task of their choosing. The scope of each case study varied (as the developers chose their own tasks); every task involved several classes while the largest task reused 23 classes. These four case studies [Holmes and Walker 2007a] aimed to address the research questions "Can industrial developers create pragmatic-reuse plans using Gilligan?" and "Do these developers perceive any benefit from planning their reuse tasks with Gilligan?"

These case studies surfaced 7 key points. (1) Industrial developers were able to successfully plan pragmatic reuse tasks for their own tasks using Gilligan. (2) 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. (3) 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: fine-grained detail could be obtained from the source code. (4) While planning a pragmatic reuse task, industrial developers are mainly interested in *outgoing* dependencies, since they want to learn how the code that they are investigating depends on the system, not how the system depends on it. (5) Industrial developers group a structural element's dependencies in two groups: direct and indirect dependencies. (6) Industrial developers think about the dependencies of a structural element more like a tree than a graph. (7) While industrial developers found creating pragmatic reuse plans beneficial, they felt that the understanding they gained from the planning process was not enough to justify its overhead in practice: they wanted the tool to help them to enact the reuse task using the decisions that they had encoded in the plan.

6.2. Second Prototype

Based on the case study feedback the planning interface was rebuilt completely for the next prototype, abandoning the visually oriented graph in favour of the current collapsible-tree text widget.

We then conducted a semi-controlled experiment to evaluate the question "Can developers more effectively locate structural dependencies using Gilligan compared to their normal practice?" [Holmes and Walker 2007b]. This experiment consisted of six experienced graduate students performing four pragmatic reuse tasks using two different treatments. These tasks involved reusing functionality from four different systems; the size of the reused code varied from 4 to 12 classes with a total of 30 to 93 dependencies.

We found 3 key points. (1) Even experienced developers are surprisingly poor at locating all 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). (2) A mechanism to alert developers to investigate dependencies that they have omitted, or to let them know that they have investigated all the dependencies, would be helpful. (3) Developers can effectively navigate structural dependencies using the second Gilligan prototype's interface. Additionally, switching the planning interface

to use standard UI widgets, instead of a graph, eased the training burden for developers as they use tree-based widgets on a daily basis.

6.3. Third Prototype

The third, and latest, prototype added the design goals supporting enactment of reuse plans and the focus on iteration. We initially wished to collect evidence about three questions [Holmes and Walker 2008]: "How much effort can semi-automating the enactment of a pragmatic-reuse plan save a developer?", "How many fewer errors must a developer manually fix when performing a pragmatic reuse task with our semi-automation?", and "How does semi-automating the enactment of pragmatic-reuse plans affect developers' productivity when they are performing pragmatic reuse tasks?"

We first conducted a case study into the ideal situation for two pragmatic reuse tasks, attempting to measure the minimum number of actions necessary to enact a plan either with standard IDE tools or through Gilligan. Two treatments for two different tasks were performed by one of the authors for this case study. The tasks involved reusing portions of 8 and 6 classes (392 and 366 lines of code respectively) from systems of 14.5 and 221 kLOC. We then conducted a controlled experiment with 8 experienced developers (4 recruited from industry) on the same two pragmatic reuse tasks as the case study. Each of the participants applied either Gilligan or standard IDE tools to one task and then switched to the other treatment for the second treatment.

A formal statistical treatment was not attempted at this point, but the evaluations strongly suggested 4 points. (1) By automating the enactment of pragmatic-reuse plans, Gilligan can considerably reduce the effort required to enact these plans. (2) 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. (3) Semi-automating the enactment of a pragmatic-reuse plan has the potential to save the developer a significant amount of time. (4) Semi-automating the enactment increases the likelihood that the developer will successfully complete a pragmatic reuse task by allowing him to focus on the problems that are important impediments to his successful completion of the reuse task.

A more complete, formal experiment with statistically valid results remained to be performed; this is described in the following section.

7. EVALUATION

While our various empirical evaluations have added to the accumulation of evidence showing that Gilligan can help developers with various aspects of these pragmatic reuse tasks, none of the evaluations have fully addressed the key question of this research: "By providing developers with a mechanism for creating and enacting pragmatic-reuse plans, can we help them to perform pragmatic reuse tasks more quickly, more accurately, and with greater confidence?"

To address this, we performed a controlled laboratory experiment. 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 test 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.

Table I. Blocks Used in the Experiment

Block Name	No. of Trials	First Task	Second Task	Third 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	
B2A	8			TD-G
B2B	8			TD-M

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

We describe our experimental design in Section 7.1, our participants in Section 7.2, the tasks they were assigned in Section 7.3, and additional details of the experimental procedure in Section 7.4. The results are described and analyzed quantitatively in Section 7.5 and qualitatively in Section 7.6. Threats to validity are discussed in Section 7.7.

7.1. Experimental Design

Our experiment was divided into two phases. The first phase was intended to address H-1 and H-2; it considered two factors: tool and task. The second phase was intended to address H-3; it considered only one factor: tool. The tool factor had two levels: using Gilligan (G) to plan and perform the pragmatic reuse task, and using only standard ("manual") IDE tools (M). The task factor had two levels, Q and RA, each of which was an independent pragmatic reuse task as described in Section 7.3. The task (TD) for the second phase did not vary. An overview of the block design is given in Table I.

Both the tool order and task order were counterbalanced in the first phase to account for confounding effects introduced by the varying skill levels of our subjects and learning effects in the experiment [Sadler and Kitchenham 1996]. 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 this context since they would already be familiar with a workable solution to the task they were repeating. Thus, each participant completed one task with Gilligan and the other with only the standard IDE tools. This design is best understood by considering the second factor to "order," resulting in a mixed design on the factors: within-(participants plus tool); and between-(participants plus order).

The first phase of the experiment consisted of 32 trials: each of our 16 participants performed two trials. The trials were assigned in four blocks B1A, B1B, B1C, B1D, each of which corresponded to a task–treatment–order tuple. Participants were randomly assigned to a block although we balanced the assignments to ensure that each of the blocks was replicated an equal number of times (4 each).

The second phase of the experiment used a between-subjects design. In this phase, the experiment consisted of 16 trials, one per participant. These trials were assigned to two blocks B2A and B2B, which specified which treatment (G or M, respectively) was used. All the trials in the second phase always happened after the trials for the first phase had been completed. We balanced the assignment to ensure that each block in the second phase was performed 8 times.

7.2. Participants

Sixteen developers participated in our study (twelve male and four female). These participants were selected from within Calgary, Canada. 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

Table II.
Overview of participants and their block assignments. The column "PRT?" records the participant's stated frequency of performing pragmatic reuse tasks. "Familiar." is familiarity and "Devel." is development.

	Role			Familiar. (1–5)		Experience (yrs.)			Blocks	
ID	UG	G	I	Eclipse	Java	Industry	Devel.	PRT?	1st	2nd
P1		✓		5	5	3	7	frequently	B1A	B2B
P2		✓		5	5	2	9	sometimes	B1A	B2A
P3		✓		5	5	–	2	sometimes	B1B	B2B
P4		✓		5	5	1	10	frequently	B1C	B2A
P5			✓	5	5	1	6	sometimes	B1D	B2B
P6			✓	5	5	2	5	sometimes	B1C	B2A
P7	✓			4	4	–	2	frequently	B1D	B2B
P8		✓		5	5	–	2	sometimes	B1B	B2A
P9		✓		4	4	1	6	rarely	B1D	B2B
P10			✓	4	4	7	13	rarely	B1C	B2A
P11			✓	5	4	3	5	frequently	B1A	B2B
P12			✓	5	4	1	5	frequently	B1B	B2A
P13	✓			5	5	–	5	sometimes	B1A	B2B
P14			✓	3	5	10	7	frequently	B1C	B2A
P15			✓	4	4	1	7	rarely	B1D	B2B
P16		✓		5	5	–	5	sometimes	B1B	B2A

levels of experience (from 1 year to 10 years), two were undergraduate students, and seven were experienced graduate students (with 2 years to 9 years of development experience). We attempted to select a sample representative of 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 participate in our experiment, each individual had to consider him- or herself "experienced" with both the Java language and with the Eclipse IDE. Ten of our participants indicated that they use Java with Eclipse on a daily basis while the remaining eight had done so at some point in the recent past. Our participants were paid $20 for their participation in our experiment. Table II provides an overview of our participants and their experience.

7.3. Tasks

The participants performed realistic tasks on deployed software systems. Each of the tasks included a description that gave the participants insight into the functionality that they were trying to reuse, why they were performing the task, and background information about the system under investigation. The participants were also pointed to a single structural element as a starting point for their investigation.

The task description was aimed to provide guidance similar to that which may be given by a supervisor or coworker 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 that the participant was employing for that trial.

The experimental tasks were from different domains and none of the tasks are similar in the functionality to be reused. By using independent tasks and systems, we aimed to reduce the possibilities of learning effects between each trial.

7.3.1. Phase 1. The two tasks involved in the first phase were chosen as "good" pragmatic reuse tasks that were sensible and could be effectively completed in the experimental time available.

7.3.1.1. 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[8] 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 given project that contained the test file and an executable jUnit[9] test suite that could check that the reused code worked as intended. Participants were pointed to one specific structural element within jGnash to begin their investigation, the `QIFParser.parseFullFile(..)` method.

jGnash consists of 34 kLOC, 404 classes, 26 interfaces, 1,405 fields, and 2,907 methods. Our successful pragmatic-reuse plan for this task[10] consisted of accepting 10 classes and interfaces, 46 fields, and 37 methods for a total of 960 LOC. We selected the jGnash QIF parser for reuse 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-level 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 task are avoided. If participants try to reuse this dependency, they become dependent upon many other irrelevant classes. We also selected this task because it is an example of a task that is 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. This expectation did not bear out.

For this task, participants were given the statement of purpose: "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."

7.3.1.2. Related artists (RA). The aTunes project[11] is a comprehensive, open-source music player and manager written in Java. One 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, participants were 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 suite that asked for related artists and checked 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 aTunes project consists of 33 kLOC, 302 classes, 17 interfaces, 1,237 fields, and 2,468 methods. Our successful reuse plan for this task consisted of accepting 10 classes and interfaces, 18 fields, and 31 methods for a total of 491 LOC. We selected the related artists feature from aTunes for reuse in the experiment as this feature represents a more difficult case than the jGnash QIF feature. The feature itself is more complicated, is spread throughout the system to a greater extent, and is colocated with many other features within aTunes. While it is colocated with other features, it is not necessarily coupled to them; the code merely *appears* to be more coupled than it actually is. In this task participants must carefully reject many structural elements that are part of the classes that the participants want to reuse but are not actually a part of the feature they want to reuse.

[8]http://jGnash.sf.net, v1.11.6.
[9]http://junit.org, v4.4.
[10]This is not necessarily a unique solution, and we provide these details solely to give a sense of the scope of the solution.
[11]http://atunes.sf.net, v1.6.0.

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

7.3.2. 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.

The third task was the main reason we split the experiment into the two phases: we were concerned that the participants might 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 that they had gained in the first phase (with both treatments) would help them to better assess the suitability of the third task.

7.3.2.1. Torrent downloader (TD). The Azureus project[12] is an open source BitTorrent peer-to-peer file transfer program written in Java. Azureus is a mature, full-featured system for transferring files using the BitTorrent protocol. 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 suite that could validate that the reused code worked correctly. The participants were pointed to one specific structural element within Azureus to begin their investigation, the method `TorrentDownloaderImpl.run()`. Azureus consists of 222 kLOC, 1,657 classes, 582 interfaces, 5,219 fields, and 12,253 methods. We did not create a reuse plan for ourselves for this task as we do not believe it is practicable to create one.

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

7.4. Experimental Procedure

The first author was physically present during each experimental session. Each participant took approximately 3 hours to complete the experiment; during this time they completed a series of discrete steps.

Orientation. We started with a short questionnaire to gain an understanding of the participants' development experience (see Table II). We then gave the participants a quick verbal overview of 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.

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

[12]http://azureus.sf.net, v2.4.0.2.

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.

Phase 1. Two trials were conducted in this phase, one for each of the tasks described in Section 7.3. The time limit for each trial was 40 minutes.

Phase 2. The "poison pill" trial was conducted in this phase. The time limit for this trial was 40 minutes. (Note that the participants were not made aware of the existence of the two phases; from their perspective, they simply performed three tasks either using Gilligan or not, as instructed.)

Debriefing. Each participant completed an exit questionnaire reflecting on their experiences for each task and the different treatments. The exit questionnaire took between 10 and 15 minutes to complete.

For each trial, we first described the task to the participant and showed the originating and target projects to him or her. The participant could ask for clarifications on the task if desired, although in the course of the experiment none of the participants asked for more detail than was already provided. We asked participants not to examine the test harness, to avoid having them find additional clues about the task; none attempted to do so. Participants were encouraged to talk aloud while they were performing their tasks so we could better understand their thought processes. Participants were able to ask us questions while they performed a task, but we only gave clarifications on the original task description.

Participants were interrupted at specific intervals during their task to answer mid-task questionnaires, at the 5-, 15-, and 40-minute marks; these questionnaires were identical in all cases. The questionnaire was not administered if they had finished a task before the checkpoint was reached. The time taken to answer the mid-task questionnaires was not deducted from the participant's total time (i.e., every participant was given the same amount of time, regardless of their level of verbosity). We used the midtask and posttask 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, as well as simulating interruptions that are known to occur frequently in industrial settings (e.g., Ko et al. [2007]).

While performing a task, if the participants decided they could not, or would not, continue this task, they were allowed to proclaim that they "gave up." After completing or aborting a task, the participants were again asked a series of questions.

7.4.1. 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 while the rest was recorded via hand-written notes (taken by the first author). No audio or video recording was performed.

In the hand-written notes, 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 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 participants 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 that the participants 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

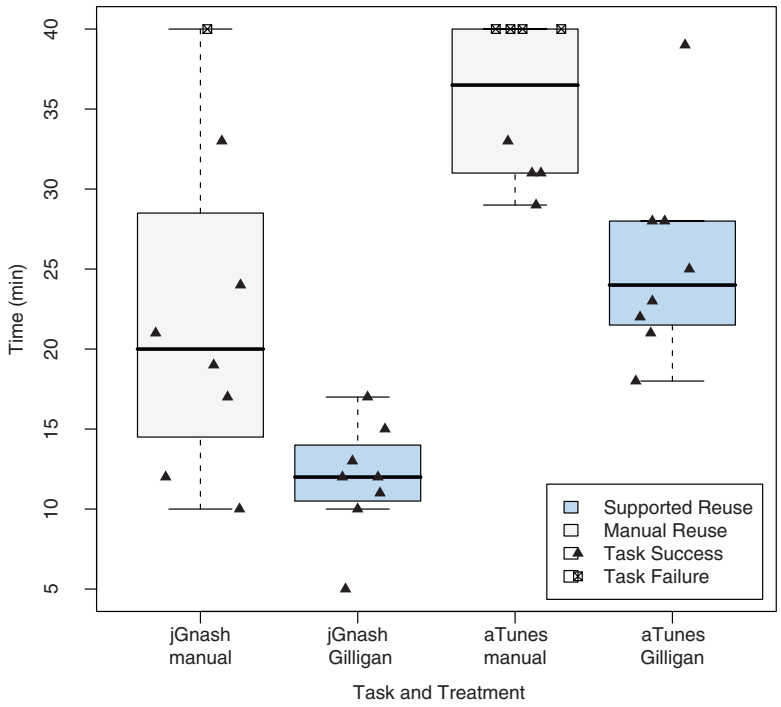

Fig. 5. Experimental block versus time to completion for the first phase.

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.

7.5. Quantitative Results

Below, we describe results in terms of our hypotheses. 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.

7.5.1. H-1 *Analysis.* To test H-1 we first examined the amount of time the participants took while performing the block B1 trials. Figure 5 shows the relationship between the experimental block and the time required for the task. 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 outside these fences can be considered outliers.

For the block B1 trials, there were only two outlier data point; these represented a participant who completed the jGnash task using Gilligan in only 5 minutes and another participant who took 39 minutes to complete the aTunes task (also with the Gilligan treatment). We did not remove the first outlier because the participant successfully completed the task: although this participant was fast, he or she managed to create a functional solution. We did not remove the second outlier to be conservative because the result biases against our approach. For five of the trials (one trial for jGnash–Manual and four trials for aTunes–Manual) the participant was unable to complete the task within the allotted 40 minutes; while none of these points were outliers, they represent

Table III. Contingency Table for Success and Failure Compared to Treatment

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

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, the bias introduced favours the manual tasks and thus the null hypothesis with respect to H-1.

We ran a repeated measures factorial ANOVA [Fisher 1918] on tool and task order. There is a significant main effect of tool ($F(1, 14) = 5.1$, $p = .04$). Participants were significantly faster using Gilligan ($M = 18.7$ minutes, $SE = 2.2$ minutes) than using standard tools ($M = 28.8$ minutes, $SE = 2.7$ minutes). The main effect of task order was not significant ($F(1, 14) = 0.9$, $p = .35$) nor was the interaction ($F(1, 14) = 0.0$, $p = .94$). Because of this, we reject the null hypothesis with respect to H-1 and consider Gilligan to be a significant influence to reduce the amount of time required to plan and perform a pragmatic reuse task.

7.5.2. H-2 *Analysis.* To test H-2 we performed a one-sided Fisher's exact test [Fisher 1922]; we chose this method over Pearson's χ^2 test [Pearson 1900; Plackett 1983] as the latter requires larger samples than we had available in this study [Chernoff and Lehmann 1954]. The contingency table used for this evaluation is given in Table III. Participants succeeded significantly more often when using Gilligan ($p = .022$). Thus we reject the null hypothesis with respect to H-2 and conclude that developers using Gilligan are more likely to successfully complete their task.

7.5.3. H-3 *Analysis.* To test H-3 we examined the amount of time it took for a participant performing task TD to abandon their task (the data is summarized in Figure 6). One participant performing the manual case gave up after only 12 minutes. This developer gave up not because he or she had reached a conclusion about its infeasibility, but because he or she had become overwhelmed during the investigation. By Chauvenet's criterion [Taylor 1996], this point can be classified as an outlier, and as a candidate for elimination from consideration. We repeated our analysis both with and without the outlier.

In the presence of the outlier, we applied the Mann-Whitney-Wilcoxon (MWW) test (a conservative, nonparametric test) [Wilcoxon 1945; Mann and Whitney 1947], as the second phase used a between-participants design, effectively reducing our sample size by half. We received a marginally significant result ($U = 16$, $p = .072$) and thus reject the null hypothesis with respect to H-3: developers using Gilligan abandon an infeasible task in less time than developers using standard IDE tools alone. The result in the absence of the outlier is stronger: ($U = 8$, $p = .012$).

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. Anecdotally, we noted that participants using Gilligan were skeptical of the task from the outset (as the first element they investigated had more than 700 transitive dependencies) and gave up once they realized they could not effectively pare down the number of dependencies. Participants performing the task manually noted the extent of the problem later on, after they had become inundated by dozens of open editors and thousands of compilation errors.

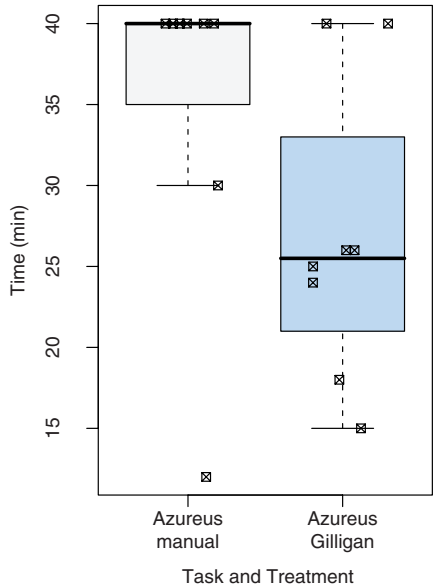

Fig. 6. Experimental block versus time to completion for the second phase.

Table IV. Number of Alternative Decisions Considered

Decision Path	# Occurrences
Accepted ⇒ Accepted	35
Accepted ⇒ Rejected	455
Rejected ⇒ Rejected	59
Rejected ⇒ Accepted	55

7.5.4. Other Quantitative Analyses. By analyzing the logs of developers using Gilligan, we gained a sense of how often they investigated alternative decisions and iterated on their reuse plan. While we were able to collect this data for the tool-supported trials (through appropriate instrumentation of Gilligan), capturing developer intent during the manual treatments was impractical. Table IV shows the number of times developers investigated alternative decisions (compared to the 2,567 decisions developers made without changing their mind); each row describes the developer's initial triaging of an element and their final one. For example, an Accepted ⇒ Accepted occurrence means the developer accepted a dependency, later tried a different decision, and ultimately decided their initial decision was correct; conversely a Accepted ⇒ Rejected occurrence indicates that the developer definitively changed their mind at some point during the task. These actions show that developers frequently investigated alternative decisions.

Developers validate or enact their reuse plan to see the effects of various decisions they have made; each of these actions serves as an implicit iteration where the developer checks the status of her task. Table V shows how many times developers iterated on their plans. The time to the first enactment and validation capture how long it took the developers to accomplish enough work that they felt they should check on their initial progress.

Both Table IV and Table V demonstrate that developers frequently iterated on their tasks, changing their minds as they gained greater insight into the source code they were reusing; they continuously explored, analyzed, and triaged the dependencies in the system after a relatively short familiarization phase.

Table V. Number of Implicit Iterations

Action	jGnash	aTunes
ENACTMENTS		
Total # of enactments	62	149
Average # of enactments	7.8	18.6
Average time to first enactment (min)	5.0	5.0
VALIDATIONS		
Total # of validations	59	114
Average # of validations	7.4	14.3
Average time to first validation (min)	2.0	5.0

7.6. Qualitative Results

During the final experiment, 254 pages of handwritten notes were taken while the participants performed their experimental tasks; these notes primarily comprise statements made by the participants but also included some observations by the experimenter. The notes were transcribed into individual comments, thoughts, or actions. The resulting 954 comments and observations were analyzed using grounded theory[13] [Corbin and Strauss 1990], which allowed us to group the comments according to their content using some coding criterion. We used an open coding approach [Miles and Huberman 1994] to assign codes to the collected data; by not predefining 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 2 and 60 comments. The second iteration shrunk the number of themes to 44 but further subdivided these into 136 subthemes; during this process many subthemes 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 comprise between 3 and 9 themes.

Three of the concept categories were *prompted* as they were a result of both observation and participant answers to specific questions that were asked of them. Five of the 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. It is important to note that all these categories were derived from observations of developers performing trials using both the Gilligan treatment and using standard IDE tools.

An overview is provided for each concept category. For each category and 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 to describe each theme; the intent of this 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. Greater detail may be found elsewhere [Holmes 2008].

7.6.1. Prompted Concept Categories. The three prompted concept categories comprise 16 themes and 66 subthemes. The first originated from questions asked of the participants before the study about pragmatic reuse tasks. The second category arose from questions asked after the study about future Gilligan improvements. The third contained task-specific observations from the study; these are not presented here as they did not provide specific insight. The participants' answers for these three categories

[13]Strictly speaking, Corbin and Strauss's [1990] grounded theory approach requires that the data be *collected* and analyzed iteratively and adaptively. Instead, merely our analysis was iterative and adaptive.

Table VI. Participants' Rationale for Performing Pragmatic Reuse Tasks

Pragmatic reuse rationale	Participants
The code needed already exists.	P2 P5 P6 P7 P10 P11 P12 P13 P14
It is faster than writing the code from scratch.	P1 P2 P4 P5 P6 P8 P11 P13 P16
To use the existing code as an exemplar.	P1 P2 P3 P4 P8 P12 P13 P14 P15
It is easier than writing the code from scratch.	P5 P7 P8 P9 P10 P12
To preserve the existing encapsulation of the existing code.	P5 P6 P14

Table VII. Impediments to Pragmatic Reuse Tasks Identified by Participants

Pragmatic reuse impediments	Participants
Uncertainty: Will it take a lot of work to reuse the code?	P1 P2 P5 P7 P10 P16
Propagating future changes can be difficult.	P4 P8 P10 P11
Keeping variable names consistent between systems.	P9 P12 P13
Risk reusing badly written code.	P4 P16
Might reuse source code you don't understand.	P7 P8

Table VIII. Gilligan User Interface Issues

Issue	Participants
Increase navigation performance.	P6 P12 P16
Provide an explicit indication of progress.	P9 P10 P13
Link validation and enactment.	P12 P15 P16
Link the editor with the structural views.	P12 P15 P16
Support incremental building.	P10 P16
Add a dependency cost recommender.	P6
Validation view is backwards.	P7 P9 P10 P12 P16
Selection behaviour is awkward.	P1 P8 P10 P11
Views can be confusing.	P3 P4 P7 P10
Greyed-out text can hide details.	P14 P16

were grouped and enumerated and are reported in more succinct form than for the organic concept categories.

7.6.1.1. Answers to questions about pragmatic reuse. At the beginning of the experiment we asked each participant about their experience with pragmatic reuse tasks. Their responses are given in Tables VI and VII.

7.6.1.2. Suggested changes to Gilligan. After having participants perform 48 experimental trials and approximately 48 hours worth of pragmatic reuse development tasks, we received a great deal of feedback about how the tool was effective and how it could be improved. These changes involve both general feature requests and specific usability fixes (see Table VIII).

7.6.2. Organic Concept Categories. The five organic concept categories that arose from the card sort 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. We next describe each category in turn.

7.6.2.1. Dependency identification. Dependency identification is a specialized form of information gathering. Participants making comments in this category 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 that 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. [16 participants for 176 comments]

(1) It is frustratingly difficult to manually identify the relevant dependencies for a pragmatic reuse task [16 for 39].
(2) Using Gilligan it is easy to locate and navigate through source code dependencies [7 for 9].
(3) It can be overwhelming and disorienting to manually navigate through unfamiliar source code [8 for 10].
(4) It is important to be able to measure and track progress while performing complex reuse tasks [15 for 40].
(5) Gilligan provides the majority of the information necessary to make informed decisions about reuse plan alternatives [14 for 29].
(6) Gilligan's validation view can help to avoid spending large amounts of time wading through lengthy lists of compilation errors [14 for 32].
(7) Being systematic aided the impression of being better able to manage large, complex tasks [7 for 15].
(8) Gilligan allowed less time to be spent in investigating irrelevant common dependencies [2 for 2].

7.6.2.2. 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 much in 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 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. [16 participants for 58 comments]

(1) Manual investigations tend to be breadth-first while Gilligan enables a more explorative methodology [4 for 5].
(2) 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 for 18].
(3) Understanding the functionality provided by the source code is key to determining whether it should be considered for reuse [6 for 11].
(4) Poorly named or modularized source code can obscure its functional role [14 for 24].

7.6.2.3. 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 that the participants 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 that the participants made while building 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. [14 participants for 74 comments]

(1) Gilligan helps to manage the complex details that arise during pragmatic reuse tasks [6 for 7].
(2) Gilligan explicitly encodes reuse decisions; this reduces overhead required to manually remember all of the salient details [9 for 16].

(3) By providing a high-level view of a reuse task, Gilligan makes it easier to get a global understanding of the source code being reused [11 for 26].
(4) Gilligan helps to focus on the reuse task rather than being overwhelmed by numerous compilation errors [5 for 6].
(5) Gilligan encourages and supports making explicit, consistent, decisions [8 for 19].

7.6.2.4. 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 participants 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 to try alternative options. 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 important on its own. [16 participants for 76 comments]

(1) Gilligan encourages and assists in investigating alternative reuse strategies [11 for 14].
(2) The manual approach frequently leads to poor decisions that are difficult to reverse [12 for 32].
(3) By helping to progress through a pragmatic reuse task one decision at a time, Gilligan enables better tracking of successes and failures [12 for 25].
(4) Gilligan increases confidence in the quality of solutions [3 for 4].

7.6.2.5. Performing pragmatic reuse tasks. Themes related strictly to performing pragmatic reuse tasks fell into this concept category. 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. [16 participants for 91 comments]

(1) Most of the manual work required during a pragmatic reuse task is conceptually simple but labour intensive [14 for 20].
(2) Reuse tasks require less time, effort, and frustration when they are performed with Gilligan [14 for 41].
(3) The manual approach encourages copying excessive amounts of code, despite the fact that developers know that this strategy is likely to lead to significant problems [7 for 12].
(4) Being able to easily remove elements that are not related to the reuse task is essential to reusing only relevant source code [8 for 18].

7.7. Threats to Validity

Internal validity. We randomized (with balancing) the assignment of subjects to experimental blocks to avoid selection bias. Different tasks on different systems while

varying treatment order avoided learning effects biasing the results. Our experimental procedure was identical regardless of the treatment being applied, and we had even expected that the manual treatment would prove more effective for small tasks. The first author took handwritten notes of all points that he deemed significant (as described in Section 7.4.1), requiring subjective judgment; however, these formed the basis solely of the qualitative results and our expectations had been that the tool would not outperform the manual approach in all cases or by so wide a margin. Thus, experimenter bias was of minimal significance.

Construct validity. The fact that the time taken by participants to answer mid-task questionnaires was not deducted from the task time means that, strictly speaking, the reported task times contain a threat to construct validity. However, the within-subjects design would have caused verbose participants to be verbose for both treatments. Informally, we can also state that the time to answer these questionnaires was small relative to the time for the task.

External validity. 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—and on a variety of scales—would be needed to gain confidence in the generalizability of these results.

Similarly, for the second phase, 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 even at this sample size we have demonstrated statistical significance for our primary hypotheses.

8. DISCUSSION

Whereas pragmatic reuse tasks have historically suffered due to the burdens they place on the developer, both in terms of program understanding and manual enactment effort, our approach has demonstrated that it can effectively improve both of these impediments. By lowering these barriers, we hope developers will be more willing to attempt larger pragmatic reuse tasks in the future. The remainder of the discussion focuses on alternative reuse strategies, shortcomings of our approach, and avenues for future research.

8.1. Alternative Reuse Strategies

Our model for pragmatic reuse involves the careful construction of a reuse plan by exploring an existing system in a bottom-up fashion. 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, developers 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, developers 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 might 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 not too much extraneous functionality is reused. (This strategy has driven our early work for recommending triage decisions [Holmes et al. 2009].)

While we have not discussed Gilligan in terms of these alternative strategies within this article, 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 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; instead, we see active attempts in industry to strip away unnecessary functionality, for example, in the Eclipse Rich Client Platform [Edgar 2003a; 2003b].

8.2. Limitations of Approach

There are four main limitations of our approach to pragmatic software reuse.

(1) *Planning overhead might overwhelm performance benefits for (very?) small reuse tasks.* A developer must expend a certain amount of effort to create a pragmatic-reuse plan; if the effort required exceeds the amount of time the developer would need to perform the reuse task manually, our approach would not make sense. As such, performing pragmatic reuse tasks using our approach for small tasks might not be worthwhile. Interestingly, while we had expected that the QIF parser task would be easier to perform manually due to the fact that the functionality was well contained, developers were much quicker using Gilligan. Nevertheless, a practical lower limit must exist; we simply have not encountered it yet. Certainly, reusing any self-contained method (e.g., no transitive dependencies) or self-contained class (e.g., none of the methods or referenced types have transitive dependencies) would be more effective using a manual approach.
(2) *Noncode artifacts are not analyzed and cannot be reused.* Our approach to planning pragmatic reuse tasks only considers the static structure of the source code. Because of this, Gilligan cannot help developers reuse documentation, configuration settings, data files, or any other infrastructure associated with a project. Research involving conceptual or other implicit dependencies could overcome this limitation.
(3) *Dynamic and data dependencies are not considered.* Our current approach relies heavily upon static analysis as the basis of our pragmatic reuse model. A richer model that captures data dependencies and runtime relationships could provide developers with more information while they are creating their plans; this could reduce the developer's dependence on compilation errors for making decisions and catch more difficult to detect relationships that may be relevant to a given task. However, we caution the community that part of the appeal of the tool is its simplicity of use; adding dataflow and dynamicity could well eliminate this simplicity.
(4) *Developers must assess task quality.* Our approach does not offer explicit recommendations to a developer about the suitability of the code they are investigating for reuse; we rely entirely on the developer's own judgment to make these decisions. While a variety of factors can contribute to the feasibility of a reuse task, we believe the most important factor is the degree to which the desired feature is dependent on the rest of its system. Often, large features that are not heavily intertwined with their systems can be far easier to reuse than small features that are tightly coupled to their system. We do not believe this to be a failing of the developer or of the tool chain; not all code can be effectively reused without expending more effort than it would take to reimplement it from scratch.

8.3. Tool Improvements and Future Research

Based on all our observations and conversations with developers throughout this research process, we have identified issues that improved tool support should consider, and several other directions that we see as useful for research into pragmatic reuse.

—*Provide an active recommendation system to help developers avoid expensive structural elements.* While observing our participants perform their pragmatic reuse tasks in the experiment, we noticed that they would frequently make "bad" decisions. While they would often notice this later, we realized that providing better information about the cost of a reuse decision would be beneficial. We have performed initial research into this issue [Holmes et al. 2009] as an extension of this current work.

—*Better integration with the IDE.* Most of Gilligan's shortcomings identified by our participants arose from usability issues whereby Gilligan's behaviour was inconsistent with that of Eclipse. For complex tools it is crucial that they behave consistently with developers' expectations to reduce frustration and to avoid having results tainted for usability reasons. While iterating on our prototypes helped, our final experiment still showed that additional improvements could increase developers' positive perception of our approach.

—*Support the identification of subtype 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 supertype. Gilligan should perform additional analysis to help developers avoid being trapped by complexities that may be obscured by the type hierarchy.

—*Implement rejection and remapping using a more sound form of transformation.* The current prototype simply comments out lines of code that contain references to rejected elements. Our evaluations found that this simplistic approach worked well enough to provide significant benefit to developers. However, it is imaginable that a more careful transformation, which replaced rejected references with nonce values and complex transformations when remapping, could provide even more benefit.

—*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. This would also help the developer to check whether assumptions in place in the original system still hold in the target system.

—*Enable changes to the original code to propagate to the reused code.* The primary negative property claimed 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. Gilligan could be augmented to provide this kind of functionality.

—*Create reusable components.* Gilligan aims to help developers to extract some functional unit that was not necessarily designed in a reusable fashion. Gilligan could be adapted to apply reuse plans for the extraction of code into reusable components, rather than integrating the source directly into the target 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.

—*Provide feedback to original authors when their code is reused pragmatically.* One of the main reasons that 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 him to understand how his code is being reused and give him data to consider when making future changes to his system.
—*Provide visualizations.* While we discarded the original, graph-based view of the earliest Gilligan prototype as being too cumbersome a means for interaction, the idea should be revisited for two reasons: (1) a graphical overview of the task could give more immediate feedback about the state of the plan; (2) a visualization of the task in the context of the larger dependency graph of the system may tell the developer more immediately where dependencies exist that would be particularly expensive to accept. The first of these ideas may be spurious in a practical setting, since the developer can view the plan in a scrollable tree view. The second of these ideas may be spurious with the provision of the dependency counts already in the tool or with the recommender mentioned above.
—*Investigate scaling and relative merits of preplanned versus pragmatic reuse.* Although we see our results as being promising for turning pragmatic reuse into a disciplined procedure, there remain significant unanswered questions about the scale and kinds of tasks that are most amenable to a pragmatic reuse approach. We expect that tool support will need to further evolve as the boundaries of applicability are probed.

9. CONCLUSION

Pragmatic reuse tasks are often stigmatized as the "wrong" way to reuse code; this stigma has developed over time due to the nonsystematic, ad hoc nature of these tasks that can lead to their undisciplined consideration and poor performance. We have presented the Gilligan tool suite for supporting pragmatic-reuse tasks. Gilligan permits the lightweight investigation of source code for possible pragmatic reuse, and simultaneously permits the construction of a pragmatic-reuse plan, as the developer makes decisions about how to treat individual dependencies in the source code. The developer can immediately see if her plan is complete, and can semi-automatically enact it, to judge its results. Gilligan eliminates the trivial, tedious steps involved in the enactment of a plan, thereby saving the developer's time and attention for the higher-level understanding needed for a successful task.

We have performed a controlled experiment with three statistically significant, quantitative results, relative to developers using standard IDE tools alone: developers using Gilligan to perform pragmatic reuse tasks involving both planning and performing the reuse task require 38% less time (despite displaying a willingness to experiment and investigate more deeply); the use of Gilligan is more likely to lead to successful pragmatic reuse tasks; and, developers using Gilligan abandon an infeasible task in less time. 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 more systematic and less risky. 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 as compared to software written from scratch.

ACKNOWLEDGMENTS

We would like to thank Brad Cossette, Rylan Cottrell, Jonathan Sillito, and the other members of the Laboratory for Software Modification Research for their support and feedback over the years; our anonymous participants for volunteering their time and energy; and our anonymous referees and our editors for their insightful comments that have led to this being better work.

REFERENCES

Baxter, I. D. 2002. DMS: Program transformations for practical scalable software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*. 48–51.

Biggerstaff, T. J. 1994. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the International Conference on Software Reuse*. 102–109.

Boehm, B. 1999. Managing software productivity and reuse. *Computer 32,* 9, 111–113.

Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

Briand, L. C., Wüst, J., and Lounis, H. 1999. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance*. 475–482.

Brooks, Jr., F. P. 1987. No silver bullet: Essence and accidents of software engineering. *Computer 20,* 4, 10–19.

Brooks, R. 1983. Towards a theory of the comprehension of computer programs. *Int. J. Man-Mach. Studies 18,* 6, 543–554.

Bruch, M., Monperrus, M., and Mezini, M. 2009. Learning from examples to improve code completion systems. In *Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations Software Engineering*. 213–222.

Burkhardt, J.-M. and Détienne, F. 1995. An empirical study of software reuse by experts in object-oriented design. In *Proceedings of the IFIP TC13 International Conference on Human–Computer Interaction*. 133–138.

Burkhardt, J.-M., Détienne, F., and Wiedenbeck, S. 2002. Object-oriented program comprehension: Effect of expertise, task and phase. *Empir. Softw. Engin. 7*, 115–156.

Caldiera, G. and Basili, V. R. 1991. Identifying and qualifying reusable software components. *Computer 24,* 2, 61–70.

Chen, Y.-F. R., Fowler, G. S., Koutsofios, E., and Wallach, R. S. 1995. Ciao: A graphical navigator for software and document repositories. In *Proceedings of the IEEE International Conference on Software Maintenance*. 66–75.

Chernoff, H. and Lehmann, E. L. 1954. The use of maximum likelihood estimates in χ^2 tests for goodness-of-fit. *Annals Math. Stat. 25,* 3, 579–586.

Corbin, J. M. and Strauss, A. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qual. Sociol. 13,* 1, 3–21.

Cordy, J. R. 2003. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the IEEE International Workshop on Program Comprehension*. 196–205.

Cordy, J. R. 2006. The TXL source transformation language. *Sci. Comput. Program. 61,* 3, 190–210.

Cottrell, R., Walker, R. J., and Denzinger, J. 2008. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 214–225.

Dahl, O.-J. and Nygaard, K. 1966. SIMULA: An ALGOL-based simulation language. *Comm. ACM 9,* 9, 671–678.

Duala-Ekoko, E. and Robillard, M. P. 2010. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Engin. Methodol. 20,*1, Article 3.

Edgar, N. 2003a. Bug 36967: Enable Eclipse to be used as a rich client platform. https://bugs.eclipse.org/bugs/show_bug.cgi?id=36967.

Edgar, N. 2003b. Eclipse Rich Client Platform UI. http://www.eclipse.org/rcp/generic_workbench_summary.html.

Eisenbarth, T., Koschke, R., and Simon, D. 2003. Locating features in source code. *IEEE Trans. Softw. Engin. 29,* 3, 210–224.

FEATHER, M. S. 1989. Reuse in the context of a transformation-based methodology. In *Software Reusability*, T. J. Biggerstaff, and A. J. Perlis, Eds., Vol. 1: Concepts and Models, Addison-Wesley, Chapter 14, 337–359.

FISCHER, G. 1987. Cognitive view of reuse and redesign. *IEEE Softw. 4,* 4, 60–72.

FISHER, R. A. 1918. The correlation between relatives on the supposition of Mendelian inheritance. *Trans. Royal Soc. Edinburgh 52*, 399–433.

FISHER, R. A. 1922. On the interpretation of χ^2 from contingency tables, and the calculation of P. *J. Royal Stat. Soc. 85,* 1, 87–94.

FRAKES, W. B. AND FOX, C. J. 1995. Sixteen questions about software reuse. *Comm. ACM 38,* 6, 75–88.

FRAKES, W. B. AND KANG, K. 2005. Software reuse research: Status and future. *IEEE Trans. Software Engin. 31,* 7, 529–536.

GAFFNEY, JR., J. E. AND CRUICKSHANK, R. D. 1992. A general economics model of software reuse. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 327–337.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA.

GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. 1995. Architectural mismatch: Why reuse is so hard. *IEEE Softw. 12,* 6, 17–26.

GARNETT, E. S. AND MARIANI, J. A. 1990. Software reclamation. *IEE/BCS Softw. Engin. J. 5,* 3, 185–191.

GOUDA, M. G. AND HERMAN, T. 1991. Adaptive programming. *IEEE Trans. Softw. Engin. 17,* 9, 911–921.

GRISWOLD, W. G. AND NOTKIN, D. 1993. Automated assistance for program restructuring. *ACM Trans. Softw. Engin. Methodol. 2,* 3, 228–269.

HOLMES, R. 2008. Pragmatic software reuse. Ph.D. thesis, University of Calgary.

HOLMES, R. AND MURPHY, G. C. 2005. Using structural context to recommend source code examples. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 117–125.

HOLMES, R., RATCHFORD, T., ROBILLARD, M. P., AND WALKER, R. J. 2009. Automatically recommending triage decisions for pragmatic reuse tasks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 397–408.

HOLMES, R. AND WALKER, R. J. 2007a. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 447–457.

HOLMES, R. AND WALKER, R. J. 2007b. Task-specific source code dependency investigation. In *Proceedings of the IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 100–107.

HOLMES, R. AND WALKER, R. J. 2008. Lightweight, semi-automated enactment of pragmatic-reuse plans. In *Proceedings of the International Conference on Software Reuse*. 330–342.

HOLMES, R.,WALKER, R. J., AND MURPHY, G. C. 2006. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Engin. 32,* 12, 952–970.

HUNT, A. AND THOMAS, D. 1999. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Section 2.7: The Evils of Duplication.

INOUE, K., YOKOMORI, R., YAMAMOTO, T., MATSUSHITA, M., AND KUSUMOTO, S. 2005. Ranking significance of software components based on use relations. *IEEE Trans. Softw. Engin. 31,* 3, 213–225.

JANSEN, S., BRINKKEMPER, S., HUNINK, I., AND DEMIR, C. 2008. Pragmatic and opportunistic reuse in innovative start-up companies. *IEEE Softw. 25,* 6, 42–49.

JANZEN, D. AND DE VOLDER, K. 2003. Navigating and querying code without getting lost. In *Proceedings of the International Conference on Aspect-Oriented Software Development*. 178–187.

JOHNSON, R. E. AND FOOTE, B. 1988. Designing reuseable [*sic*] classes. *J. Obj.-Oriented Program. 1,* 2, 22–35.

KAPSER, C. AND GODFREY, M. W. 2008. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empir. Softw. Engin. 13,* 6, 645–692.

KERSTEN, M. AND MURPHY, G. C. 2005. Mylar: A degree-of-interest model for IDEs. In *Proceedings of the International Conference on Aspect-Oriented Software Devel.* 159–168.

KIM, M. AND NOTKIN, D. 2006. Program element matching for multi-version program analyses. In *Proceedings of the International Workshop on Mining Software Repositories*. 58–64.

KIM, M., SAZAWAL, V., NOTKIN, D., AND MURPHY, G. 2005. An empirical study of code clone genealogies. In *Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations Software Engineering*. 187–196.

KO, A. J., DELINE, R., AND VENOLIA, G. 2007. Information needs in collocated software development teams. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 344–353.

KOENEMANN, J. AND ROBERTSON, S. P. 1991. Expert problem solving strategies for program comprehension. In *Proceedings of the ACM SIGCHI Conference on Human Factors Computing Systems.* 125–130.

KRUEGER, C. W. 1992. Software reuse. *ACM Comput. Surveys 24,* 2, 131–183.

KRUEGER, C. W. 2000. Software product line reuse in practice. In *Proceedings of the IEEE Symposium on Application-Specific Systems Software Engineering Technology*. 117.

LANGE, B. M. AND MOHER, T. G. 1989. Some strategies of reuse in an object-oriented programming environment. In *Proceedings of the ACM SIGCHI Conference on Human Factors Computing Systems.* 69–73.

LANUBILE, F. AND VISAGGIO, G. 1997. Extracting reusable functions by ow graph-based program slicing. *IEEE Trans. Softw. Engin. 23,* 4, 246–259.

LAWRANCE, J., BELLAMY, R., BURNETT, M., AND RECTOR, K. 2008. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *Proceedings of the ACM SIGCHI Conference on Human Factors Computing Systems*. 1323–1332.

LEMOS, O. A. L., BAJRACHARYA, S., OSSHER, J., MASIERO, P. C., AND LOPES, C. 2009. Applying test-driven code search to the reuse of auxiliary functionality. In *Proceedings of the ACM Symposium on Applied Computing*. 476–482.

LETOVSKY, S. 1986. Cognitive processes in program comprehension. In *Proceedings of the Workshop on Empirical Studies of Programmers*. 58–79.

LIONS, J.-L., LÜBECK, L., FAUQUEMBERGUE, J.-L., KAHN, G., KUBBAT, W., LEVEDAG, S., MAZZINI, L., MERLE, D., AND O'HALLORAN, C. 1996. Ariane 5: Flight 501 failure. Tech. rep., Ariane 501 Inquiry Board.

LITTMAN, D., PINTO, J., LETOVSKY, S., AND SOLOWAY, E. 1986. Mental models and software maintenance. In *Proceedings of the Workshop on Empirical Studies of Programmers*. 80–98.

MANN, H. B. AND WHITNEY, D. R. 1947. On a test of whether one of two random variables is stochastically larger than the other. *Annals Math. Stat. 18,* 1, 50–60.

MCILROY, D. 1968. Mass-produced software components. In *Software Engineering: Report on a Conference on by the NATO Science Committee*. 138–155.

MILES, M. B. AND HUBERMAN, M. 1994. *Qualitative Data Analysis: An Expanded Sourcebook (2nd Ed.)*. Sage Publications, Inc., Thousand Oaks, CA.

MORISIO, M., EZRAN, M., AND TULLY, C. 2002. Success and failure factors in software reuse. *IEEE Trans. Softw. Engin. 28,* 4, 340–357.

MÜLLER, H. A. AND KLASHINSKY, K. 1988. Rigi: A system for programming-in-the-large. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 80–86.

NCUBE, C., OBERNDORF, P., AND KARK, A. W. 2008. Opportunistic software systems development: Making systems from what's available. *IEEE Softw. 25,* 6, 38–41.

NORMAN, D. 1998. *The Design of Everyday Things*. MIT Press.

OPDYKE, W. F. 1992. Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign.

PARNAS, D. L. 1976. On the design and development of program families. *IEEE Trans. Softw. Engin. 2,* 1, 1–9.

PARSONS, J. AND SAUNDERS, C. 2004. Cognitive heuristics in software engineering: Applying and extending anchoring and adjustment to artifact reuse. *IEEE Trans. Softw. Engin. 30,* 12, 873–888.

PEARSON, K. 1900. Mathematical contributions to the theory of evolution VII: On the correlation of characters not quantitatively measurable. *Philos. Trans. Royal Soc. London, Series A 195*, 1–47 & 405.

PENNINGTON, N. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychol. 19,* 3, 295–341.

PLACKETT, R. 1983. Karl Pearson and the chi-squared test. *Int. Statist. Rev. 51,* 1, 59–72.

POULIN, J. S., CARUSO, J. M., AND HANCOCK, D. R. 1993. The business case for software reuse. *IBM Syst. J. 32,* 4, 567–594.

PRIETO-DÍAZ, R. 1993. Status report: Software reusability. *IEEE Softw. 10,* 3, 61–66.

RAVICHANDRAN, T. AND ROTHENBERGER, M. A. 2003. Software reuse strategies and component markets. *Comm. ACM 46,* 8, 109–114.

REISS, S. P. 2009. Semantics-based code search. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*. 243–253.

ROBILLARD, M., WALKER, R., AND ZIMMERMANN, T. 2010. Recommendation systems for software engineering. *IEEE Softw. 27,* 4, 80–86.

ROBILLARD, M. P. 2008. Topology analysis of software dependencies. *ACM Trans. Softw. Engin. Methodol. 17,* 4, Article 18.

ROBILLARD, M. P. AND MURPHY, G. C. 2007. Representing concerns in source code. *ACM Trans. Softw. Engin. Methodol. 16,* 1, 3.

ROSSON, M. B. AND CARROLL, J. M. 1996. The reuse of uses in Smalltalk programming. *ACM Trans. Comput.-Hum. Interact. 3,* 3, 219–253.

ROUET, J.-F., DELEUZE-DORDRON, C., AND BISSERET, A. 1995. Documentation as part of design: Exploratory _eld studies. In *Proceedings of the IFIP TC13 International Conference on Human–Computer Interaction.* 213–216.

SADLER, C. AND KITCHENHAM, B. A. 1996. Evaluating software engineering methods and tools| Part 4: The inuence of human factors. *SIGSOFT Softw. Engin. Notes 21,* 5, 11–13.

SELBY, R. W. 2005. Enabling reuse-based software development of large-scale systems. *IEEE Trans. Softw. Engin. 31,* 6, 495–510.

SEN, A. 1997. The role of opportunism in the software design reuse process. *IEEE Trans. Softw. Engin. 23,* 7, 418–436.

SHNEIDERMAN, B. 1980. *Software Psychology: Human Factors in Computer and Information Systems*. Little, Brown, and Co.

SILLITO, J., MURPHY, G. C., AND DE VOLDER, K. 2008. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Engin. 34,* 4, 434–451.

SNYDER, A. 1986. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 38–45.

SOLOWAY, E. AND EHRLICH, K. 1984. Empirical studies of programming knowledge. *IEEE Trans. Softw. Engin. 10,* 5, 595–608.

SOLOWAY, E., LAMPERT, R., LETOVSKY, S., LITTMAN, D., AND PINTO, J. 1988. Designing documentation to compensate for delocalized plans. *Comm. ACM 31,* 11, 1259–1267.

STANDISH, T. A. 1984. An essay on software reuse. *IEEE Trans. Softw. Engin. 10,* 5, 494–497.

STOREY, M.-A. D., WONG, K., AND MÜLLER, H. A. 2000. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program. 36,* 2-3, 183–207.

SZYPERSKI, C. 2002. *Component Software: Beyond Object-Oriented Programming*. ACM.

TAYLOR, J. R. 1996. *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements* 2nd Ed. University Science Books.

TOOMIM, M., BEGEL, A., AND GRAHAM, S. L. 2004. Managing duplicated code with linked editing. In *Proceedings of the IEEE Symposium on Visual Lang. Human-Centric Computing*. 173–180.

TRACZ, W. 1990. Where does reuse start? *SIGSOFT Softw. Engin. Notes 15,* 2, 42–46.

VAN GURP, J. AND BOSCH, J. 2002. Design erosion: Problems and causes. *J. Syst. Softw. 61,* 2, 105–119.

VANCIU, R. AND RAJLICH, V. 2010. Hidden dependencies in software systems. In *Proceedings of the IEEE International Conference on Software Maintenance*.

VON MAYRHAUSER, A. AND VANS, A. M. 1995. Program comprehension during software maintenance and evolution. *Computer 28*, 44–55.

WILCOXON, F. 1945. Individual comparisons by ranking methods. *Biometrics Bull. 1*, 80–83.

YELLIN, D. M. AND STROM, R. E. 1997. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst. 19,* 2, 292–333.

Received February 2010; revised November 2010, March 2011; accepted April 2011