Automatically Recommending Triage Decisions for Pragmatic Reuse Tasks

Reid Holmes
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA, USA
Email: rtholmes@cs.washington.edu

Tristan Ratchford, Martin P. Robillard
School of Computer Science
McGill University
Montreal, QC, Canada
Email: tratch, martin@cs.mcgill.ca

Robert J. Walker
Dept. of Computer Science
University of Calgary
Calgary, AB, Canada
Email: walker@ucalgary.ca

***Abstract*—Planning a complex software modification task imposes a high cognitive burden on developers, who must juggle navigating the software, understanding what they see with respect to their task, and deciding how their task should be performed given what they have discovered. Pragmatic reuse tasks, where source code is reused in a white-box fashion, is an example of a complex and error-prone modification task: the developer must plan out which portions of a system to reuse, extract the code, and integrate it into their own system. In this paper we present a recommendation system that automates some aspects of the planning process undertaken by developers during pragmatic reuse tasks. In a retroactive evaluation, we demonstrate that our technique was able to provide the correct recommendation 64% of the time and was incorrect 25% of the time. Our case study suggests that developer investigative behaviour is positively influenced by the use of the recommendation system.**

***Keywords*—pragmatic software reuse tasks; triage decisions; recommendation systems; structural relevance; cost; source code analysis; retroactive evaluation.**

I. INTRODUCTION

In industrial practice, *pragmatic reuse tasks* arise when a developer wants to reuse existing functionality that was not designed to be reused as needed [1]. When exploring the source code they want to reuse, developers balance two competing concerns: the desire to reuse as much code as possible to obtain the needed functionality for their task, and the desire to eliminate as much irrelevant reused code as practical. This latter restriction is essential, as every piece of code is dependent on other source code elements; if the developer is not careful, they may unwittingly add spurious functionality that will increase maintenance costs or can cause problems in their system. Unfortunately, it is generally difficult for developers to determine the relevance and cost of reusing a piece of code through simple inspection, and then to make an informed decision about the tradeoff involved.

As developers navigate through the source code, they create a plan of their reuse activity, either explicitly or mentally: *What elements should I reuse? Should I truncate this dependency? What is the cost of reusing this element?* The cost associated with reusing an element is related to the number of elements that may also need to be brought over to the target system because the element is dependent upon them. Unfortunately, creating these plans places a high cognitive burden on the developer; this increases the likelihood of making a mistake, either by reusing an expensive dependency without realizing it, or rejecting an inexpensive dependency that was relevant to their task.

Previous work provides an environment (called Gilligan) for supporting the planning [1] and performance [2] of pragmatic reuse tasks. In a previous study [3], it was shown that Gilligan can result in the faster implementation of a reuse plan and a higher rate of success. However, Gilligan still leaves significant room for improvement: developers often make poor decisions about the structural relevance and reuse cost of individual program elements as they create their plan, although they generally recognize them eventually. These poor decisions cause developers to waste time investigating infeasible reuse paths and confound their understanding of the system.

Our goal in this paper is to automatically provide recommendations to developers as they create their reuse plan about the relevance and cost of any element they are investigating. These recommendations are also adaptive: as the developer triages elements in their reuse plan, the recommender adjusts its notion of which elements are relevant to the task. Reliably creating these kinds of recommendations can be difficult because developers decide about the elements they want to reuse not just in terms of cost and structural relevance, but also in terms of other factors such as semantics and perceived code quality.

The recommendation system that we have incorporated into the Gilligan environment is based on two measures: structural relevance and cost of reuse. We extend the Suade structural relevance algorithm [4] to operate within the context of pragmatic reuse decisions, and integrate it into Gilligan. The cost-of-reuse measure is based on the shape of the graph of depended-upon descendants of an element: few nearby descendants means that the cost will be relatively low; an exponential decay model is used to represent this intuition.

We evaluated our approach by replaying the sessions of 16 developers as they performed two pragmatic reuse tasks using Gilligan in a controlled experiment. We generated a recommendation for each decision they made during their

task, at the instant before they made it, to see what support we would have given them for that decision. In addition, we undertook an informal case study to observe whether the recommender was likely to affect developer behaviour, either positively or negatively.

Section II outlines the existing Gilligan environment and illustrates a scenario that would benefit from greater support. Related work is discussed in Section III. Our approach is detailed in Section IV. We evaluate our approach in Section V. Section VI discusses other issues.

This paper contributes an automated recommendation approach for pragmatic reuse triage decisions, combining an existing structural relevance algorithm with a novel, locally-weighted fan-out measure to model cost of reuse.

II. Motivation

In this section we describe the Gilligan environment for pragmatic reuse tasks, outline a scenario in which a developer uses Gilligan for such a task, and describe the resulting issues and others that we have observed.

A. Gilligan Environment

The Gilligan environment helps developers to plan and perform pragmatic reuse tasks [1]–[3]. Developers investigate the source code they are considering for reuse by using a tree-based abstraction of the fan-out portion of the program dependency graph; from any source code element that is investigated, the elements upon which that element depends, either immediately or transitively, must be dealt with to ensure that any required elements are reused while irrelevant elements are eliminated. Gilligan surfaces these depended-upon elements through static analysis, saving the developer from having to navigate through, and reason about, many different source code files while trying to identify the relevant and irrelevant elements.

As developers investigate source code elements, they make triage decisions by deciding to *accept* ("I want to reuse this element") or *reject* ("I don't want to reuse this element") each element (a third option exists—*remap* ("I have similar functionality I want to use instead of reusing this element")—but we do not consider it in this current work; see Section VI). Gilligan also automatically triages elements that are *common* between the source and target systems; this occurs when libraries or frameworks are shared between the systems. Rejecting (or remapping) an element causes its descendants to be rejected implicitly when they become otherwise unreachable. Triage decisions are stored as a *pragmatic-reuse plan*. This plan can be mostly automatically enacted; that is, Gilligan takes the plan and automatically copies the code from the source system, modifies it, and integrates it into the developer's target system. This relieves the developer of much of the manual effort associated with performing these tasks.

B. Scenario

Consider a developer undertaking the task of parsing a file encoded in the Quicken Interchange Format (QIF). Rather than starting from scratch, they choose to search for existing QIF parsers online; the developer finds the Java-based jGnash[1] project, which seems to meet their needs.

The developer starts their reuse task by searching through the jGnash source code to find a starting point for their investigation; this yields `QifParser.parseFullFile(File)`. Selecting this element and starting the Gilligan reuse environment, the developer begins their investigation.

Gilligan provides an investigative view with three panes, each of which consists of a tree that has multiple columns of additional information for each element in the tree. Figure 1 shows the developer's initial view for this task after selecting `parseFullFile(File)`. The leftmost pane shows the list of elements that the developer has triaged or is investigating; the middle pane shows the direct dependencies of the developer's selection; the rightmost pane shows the transitive closure of dependencies of the selection. Three additional columns within each pane show the decision the developer has made (as a colour), and two numbers, one enumerating the number of direct dependencies and the other enumerating the number of dependencies in the transitive closure.

As the developer navigates through the dependencies in the system, they triage each one by deciding whether to reuse it (annotated in green), or to reject it (annotated in red). At any time they can press a single button to enact the plan wherein the accepted elements are copied to their system and automatically modified to resolve a large proportion of the resulting compilation errors from this copy operation. After triaging dependencies for 15 minutes the developer realizes that they have navigated down into a myriad of dependencies in the `jgnash.engine` package and feels like they are off-track. After a few minutes they realize that their acceptance of the `QifTransaction._account` field, of type `Engine`, led them down this path. By scrutinizing `_account` they realize that the code they are reusing only uses the field once, and in a way that is irrelevant to their task; they decide to reject the field rather than reuse the dozens of irrelevant classes it transitively depends on.

Identifying that `_account` should have been rejected was difficult for the developer for two reasons: first, they could not tell that the field was lightly used by the code they were reusing; second, they could not tell the transitive burden of the field—in `QifTransaction` it simply looks like another one-line field.

C. Issues

Gilligan surfaces elements to the developer for consideration, and makes it clear what has and has not been triaged.

[1] http://jgnash.sf.net

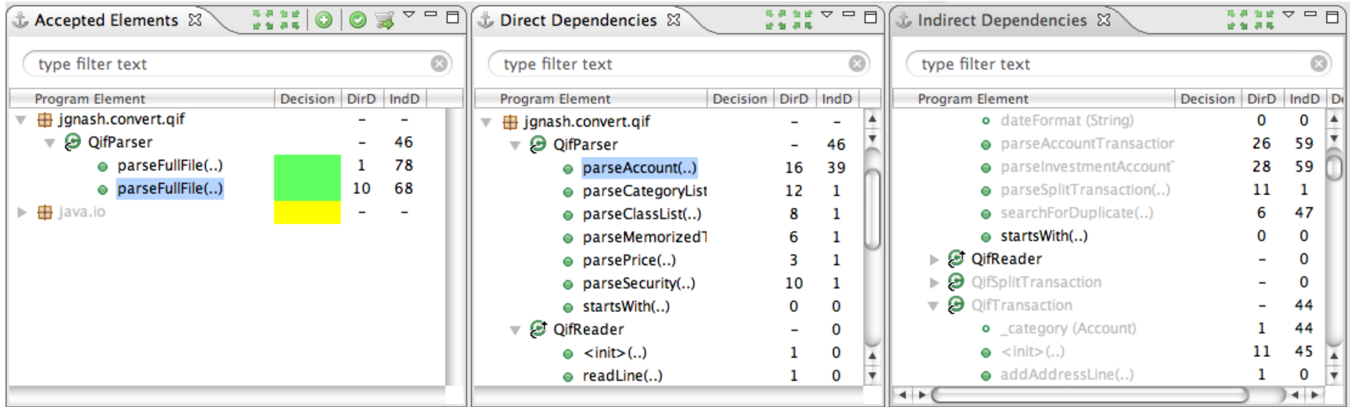

Figure 1. The Gilligan environment.

We have previously observed [3] that, while Gilligan can significantly increase the speed and quality with which a developer can perform a pragmatic reuse task, developers still take a long time to make triage decisions and frequently make incorrect decisions that they later must revisit and revise. Gilligan provides two simple indications of the cost of reusing an element: the count of immediately depended-upon elements and the count of elements in the transitive closure of dependencies. Several industrial developers who have used Gilligan have requested that the tool provide explicit advice about how a given element should be triaged, and we have found that the two counts do not implicitly help much in this regard. Something more is needed.

III. Related Work

Software reuse has long been advocated as a mechanism to reduce development time, increase developer productivity, and decrease defect density [5]–[9]. Subsequent studies have been performed to provide initial validation of such claims [10]–[13].

The act of reusing source code that was not designed to be reused as needed, in a particular context, has been known by many monikers: code scavenging [7], ad hoc reuse [14], opportunistic reuse [15], and copy-and-paste reuse (also known as cut-and-paste or copy-and-modify) [16]. We choose the term *pragmatic reuse* because each of the above monikers carry negative connotations, while we believe that pragmatic reuse tasks can be appropriate and effective [17]–[19]. Indeed, non-black-box reuse tasks are not atypical; Selby et al. [20] found that 47% of the reused source code within NASA—an organization committed to reuse—was reused in a non-black-box manner.

Many of the problems impeding the successful reuse of software are cognitive in nature [21]. In order to overcome these cognitive impediments, Fischer [21] argues that more information is not needed, but that the information that is currently available must be structured more effectively. He further argues that tools must support the developer in safely investigating alternative reuse scenarios. Gilligan and our present attempt at extending it with a recommendation system provide precisely this support.

Krueger [7] states, "In practice, the overall effectiveness of [pragmatic software reuse] is severely restricted by its informality". Frakes and Kang [22] identify two impediments to pragmatic reuse tasks: first, they state that development tools may not be effective at promoting reuse; second, they note that the lack of process associated with these hampers reuse efforts. Other researchers have identified similar issues [23]–[25]; the previous work on Gilligan [1]–[3] directly addresses these.

Cordy [17] provides many reasons why industrial organizations reuse code via clones instead of refactoring the code base; 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 their subsystems better. Toomim et al. [18] argue that there are cognitive costs associated with abstraction and that copy-and-paste development provides a mechanism to avoid some of these costs. Such approaches are designed to address the issue of copying code *within* a single system; instead, our primary intent is to enable developers to reuse large-scale functionality from external systems more effectively.

A number of systems [26]–[28] extract reusable code based on structural factors, but without consideration of the specific task to be supported.

Garlan et al. [29] point out some of the major hurdles involved in pragmatic reuse tasks; in particular, the fact that there can be deep conflicts between the code to be reused and the target system where it is to be reused. Such "architectural mismatches" point to why attempts to simply reuse the transitive closure of dependencies from a key starting point, will not always lead to successful reuse tasks. CodeGenie [30] extracts code slices from existing systems based on test cases selected by the developer; slicing

amounts to extracting the transitive closure of dependencies. In addition, dependencies only tell us how some starting point interacts with the rest of the original system; some of these dependencies may simply be irrelevant in the target system, and should be removed when practicable to avoid long-term costs from maintenance.

The FEAT tool [31] helps developers to create descriptions of scattered software, called concerns. A concern in FEAT is a graph where the nodes are software elements and the edges are the relationships between them. A pragmatic-reuse plan (see Section II) is also a graph, but extends the nodes to include triaging information that capture how the structural element should be managed when the pragmatic reuse task is performed.

IV. Approach

Our approach applies software dependency analysis in a novel and dedicated context: the recommendation of software elements that can be pragmatically reused in a white-box fashion. To achieve this, we leverage an existing recommendation algorithm [4] for structural relevance of a given programmatic element (Section IV-A), modified to fit our context (Section IV-B); and we develop a measure for estimating the structural cost of reusing that element (Section IV-C). A tension exists between the relevance of an element and its cost; we provide a general model for the combination of these two factors in order to arrive at recommendations about whether or not to reuse an element (Section IV-D). This general model is instantiated and evaluated in Section V.

A. Background: Structural Relevance

Topology analysis of software dependencies is a technique to produce recommendations for software navigation based on the topology of a software dependency graph [4]. This technique was implemented in a tool called Suade. Although at least three structure-based recommendation tools have been described in the literature [4], [32], [33], the Suade approach was a good candidate for our application due to its ready availability, support for Java, and most of all, our extensive experience with it. The following summary of this technique is adapted from previous work [34].

The idea behind Suade's algorithm is to rank elements based on the closeness of their structural association with program elements in a set of interest that represents a developer's context of investigation (e.g., all the fields and methods related to a change task).

The Suade algorithm assumes the presence of a set of elements, the *interest set*, known to be related to a task. It then analyzes the topology of the structural dependency graph that includes the elements from the interest set and their immediate structural neighbours (e.g., callers, callees, etc.), to rank all the structural neighbours in order of estimated likelihood to also be of interest. The algorithm takes into account two heuristics involving the structural relationships between elements: *specificity* and *reinforcement*. Specificity evaluates the "uniqueness" of a dependency between a given element and its structural neighbours. For example, imagine that the element x in the interest set $\{x, y\}$ calls five other elements, x_1 through x_5, and that y is called by two other elements, y_1 and y_2; then y_1 and y_2 are considered more specific to the interest set than x_1 through x_5. Elements with higher specificity are ranked higher than those with lower specificity.

Reinforcement evaluates the strength of the intersection between the interest set and the structural dependents of a given element. For example, imagine that the interest element x is related to five other elements x_1 through x_5, and that four of these elements—x_1 through x_4—are already in the interest set; then the remaining element (x_5) is considered heavily reinforced. On the other hand, if none of the dependents are also in the interest set, the algorithm does not consider the elements to be reinforced. Elements with higher reinforcement are ranked higher than those with lower reinforcement.

The algorithm works by separately analyzing the "calls", "called by", "accesses" and "accessed by" relations. First, it obtains, for each element in the interest set, all elements related to it by the relation type currently analyzed. For example, for the relation type "called by", it obtains all callers of each method in the interest set. A formula is then used to produce, for each related element, a degree of potential interest for the element that is based on the specificity and reinforcement heuristics. The results of the analysis of each relation are then merged. In the end, the algorithm produces a single ordered set of elements directly related to the interest set.

B. Merging Suade with Gilligan

Pragmatic-reuse plans consist of a set of structurally related elements that provide some piece of functionality that a developer wants to reuse. As such, using Suade to provide the developer with a measure of structural relevance could be beneficial because it can give insight to the importance of an element. The recommendations provided by Suade are also adaptive; Suade adapts its recommendations as the developer triages additional elements.

Suade's algorithm generates recommendations for all elements in the program dependency graph (comprised of method call and field reference relationships) that are related to elements in the interest set; it also leverages the transpose relationships within the program dependency graph, so both "x depends on y" and "y is depended upon by x" relations are considered in the analysis. In contrast, Gilligan does not consider transpose relations; only outgoing dependencies are displayed in the Gilligan user interface. Gilligan also considers inheritance and "uses" relationships, which Suade does not. A "uses" relationship will arise when a type is

referenced by a method (e.g., as a parameter or return type), but a specific method call or field access might not be made on that type. Additionally, fields have a uses relationship on their type.

Suade represents structural relevance as a normalized value between 0 and 1. If an element is not related to the interest set, Suade does not recommend it. As such, Gilligan will display a value of "n/a" beside it to indicate that it is not structurally related to the context set. Similarly, since Suade is generating recommendations for the inclusion of potential elements, it does not assign a degree to any element already in the interest set.

Merging the two algorithms required three modifications to the Suade algorithm.

1. Rejected elements: We consider the triaged elements in Gilligan to form the interest set for Suade; accepted and rejected elements are then treated differently for Suade's analysis. An element within the transitive closure of any accepted element will be assigned a degree as per the standard implementation of Suade; an element within the transitive closure of any rejected element will be assigned a degree of 0. The intuition for this design is that any element that is structurally related to a rejected element will also not be pertinent to the context. If an element lies in the transitive closure of more than one element, the highest resulting degree is assigned to it.

2. Constants: We modified Suade such that, if a method accesses a constant field, that field will have a degree at least as high as the method that accessed it. The intuition behind this is that constants are often an essential part of a method's functionality and thus can be thought of as an extension of the method itself.

3. Types: The third modification entailed giving a structural relevance degree to each type, based on how heavily its members were depended on by the interest set. This modification was added to give users a feel for how connected a particular type is to the interest set. A type's degree is equal to the average of all the degrees of its members. Only types with members that are structurally related to the interest set will receive a degree. If this condition is met, elements that are not structurally related to the interest set but reside in the type, are considered to have a degree of 0 in this calculation. If an interest element resides in the type, it is considered to have a degree of 1 in this calculation.

Through the integration with Suade, we have reduced one of Gilligan's analytic shortcomings: Gilligan provides a tree-view of the reuse plan for simplicity and ease of navigation for the developer [35]; unfortunately, this can obscure the fact that some elements are more relevant to a reuse task than others (e.g., a method that is called multiple times from several different locations) that may otherwise be visible by examining a graph of the program's structure. While the awareness of structural relevance is necessary for evaluating these tasks, it is not sufficient as sometimes the cost of a dependency may overwhelm its structural relevance; to help identify these situations, we next introduce a means to capture the cost of a structural element.

C. Reuse Cost Measure

Equation 1 defines the cost of accepting an element x in terms of the number of its descendants, weighted by an exponential decay function of distance from x.

$$\mathrm{cost}(x) = \sum_{d=1}^{d_{\max}} |\mathrm{Desc}(x, d)| \cdot e^{\alpha(d-1)} \qquad (1)$$

The variable d represents the distance of a descendant from x (i.e., the length of the shortest path from x to the descendant) and α represents a decay constant. We define $\mathrm{Desc}(x, d)$ as the set of descendants of x at distance d from x. For example, if method `a()` calls methods `m1()` through `m8()`, references the type `String`, and calls `length()` on `String`, $|\mathrm{Desc}(\texttt{a()}, 1)|$ would equal 10. Also it should be noted that all members of the type `String` will be in $\mathrm{Desc}(\texttt{a()}, 2)$, except `length()`.

The intuition behind this is that there is an inherent cost whenever an element is accepted into a pragmatic reuse plan. Elements with many nearby dependencies require the user to invest more of their time investigating these elements and fixing dependency conflicts, than for an element that has few or mostly distant dependencies. The likelihood that distant elements will ever be triaged is far lower than nearby ones.

The cost measure is represented by an unbounded non-negative real, although in practice we have found that this number tends to usually remain below 30.

1) Calibrating the cost measure: To calibrate the cost measure, we created a set of scenarios concerning patterns of fan-out from elements to their descendants and subjectively estimated how we thought the cost should be reported for each (see Table I). These scenarios derive solely from our general experiences with fan-out and how importantly one

Table I
SCENARIOS USED TO CALIBRATE THE COST MEASURE. THE "DISTANCE" COLUMNS INDICATE THE NUMBER OF CHILDREN AT THAT DISTANCE FROM THE ROOT (WHICH HAS DISTANCE 0 FROM ITSELF). "COST" IS A SUBJECTIVE ESTIMATE OF WHAT THE RESULTING COST OF REUSING THE ROOT SHOULD BE.

Case	**Distance**							**Cost**
	1	2	3	4	5	6	7	
a	1	12	11	33	16	4	0	3–7
b	37	1	1	1	1	1	1	37.1–37.5
c	11	1	27	1	1	1	1	11.5–12.5
d	1	1	1	1	3	9	27	1.1–1.5
e	1	1	1	1	9	3	27	1.1–1.5
f	2	4	8	16	13	0	0	4–20
g	2	4	8	0	0	0	0	2.5–5
h	43	0	0	0	0	0	0	43
i	20	1	1	1	20	0	0	20.2–20.5

needs to avoid traversing the dependency path in some situations. For example, in Case b, the root possesses 37 children, but only one of these has a long chain of single descendants; thus, one would have to pay for the immediate cost of the children, but moving beyond that level would involve little cost, and we estimated the total cost of accepting the root as a little over 37 elements. Conversely, a fan-out with an exponential growth pattern (like Case f) will likely have a much higher cost, but when exponential growth does not begin until some distant level of descendant (like Cases d and e) the impact of the exponential growth does not imply an immediate cost.

The result of this informal investigation led us to choose α equal to –1.5 as a reasonable value. No formal regression was conducted, since the estimated costs are subjective and represent ranges. Further, objective calibration was needed before the cost measure could be used within the recommender (see Section V-A), so this informal procedure sufficed.

D. Combining Cost with Relevance

Structural relevance and reuse cost are largely independent measures, both from a philosophical perspective and from our empirical investigations. As such, our recommendation model combines them by considering the space that they describe as orthogonal dimensions. Since our goal is to more effectively focus the developer's attention on elements that will require careful attention, we chose to produce three-valued recommendations: for some parts of the space, it recommends acceptance; for others, rejection; and for others, it cannot confidently accept or reject and so yields "no recommendation" (or equivalently, "none").

Roughly, the space is divided into regions for recommendation as illustrated in Figure 2; Section V-A describes how we calibrate the locations (slopes and intercepts) of the lines indicated. We use straight lines for simplicity and because it is not apparent that more sophisticated curves would improve our results. Our intuition for these regions is as follows. Elements with very low cost ought to be recommended for acceptance, as the time spent in carefully considering them would not lead to significant savings (defining "very low" is left to calibration). Elements with structural relevance measures of 0 or "n/a" should likely be treated specially. Elements with sufficiently high structural relevance but low cost ought to be recommended for acceptance, since it is likely that these are pertinent to the task, and the amount of irrelevant code that will be brought in is expected to be small. Elements with sufficiently low structural relevance but high cost ought to be recommended for rejection, as they are unlikely to be pertinent and will entail the inclusion of a sizeable amount of irrelevant code if accepted. For the region in between, it is unclear whether accepting or rejecting is truly warranted. These half-planes defining the regions overlap, and determining which half-plane should take precedence in the overlap is a matter for calibration.

The general model for the recommendation system is a set of three *regions*, R_{accept}, R_{reject}, and R_{none}, where each region R_i consists of a unique (not necessarily connected) portion of the plane spanned by the cost measure and the structural relevance. Each region also defines a recommendation r_i if the element falls within it ($r_i = i$). The three regions must not overlap and they must cover the plane completely; thus, any two fully-specified regions will imply the third. An illustration of the idea is presented in Figure 2.

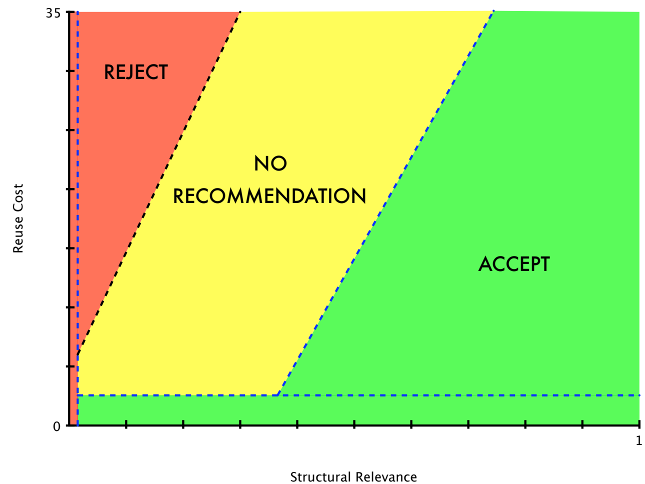

Figure 2. The conceptual recommendation regions in the space of structural relevance versus reuse cost.

V. Evaluation

To evaluate our approach, we first calibrate our recommendation model against a few data sets (Section V-A); we then retroactively apply the calibrated model to recorded developer sessions to see how the recommendations could have helped them for these tasks (Section V-B). Finally, we perform the experimental tasks ourselves again, using the recommender, to see how having the recommendations changes how we perform the tasks (Section V-C).

A. Calibrating the Recommender

We use the decision logs for three successful pragmatic reuse plans we performed in the past to calibrate the recommender: (1) reusing a charting component from Azureus[2]; (2) reusing the NMEA parser from the Gpsylon project[3]; and (3) reusing the lines of code counter from the Metrics project[4]. Each of these tasks involved reusing more than 400 LOC from at least 8 classes.

[2]http://azureus.sf.net v2.4.0.2
[3]http://gpsmap.sf.net v0.5.2
[4]http://metrics.sf.net v1.3.6

Each region of the recommendation model can have an arbitrary boundary. Consider the straight lines that we have used (one vertical line, one horizontal line, and two arbitrary lines): there are 6 independent parameters in defining them, plus the order of precedence of their defined half-planes is also essentially independent, yielding up to an additional 24 permutations. As a result, even with the simple boundaries under consideration, we are dealing with over 100 degrees of freedom. Attempting to formally optimize the general model would be a daunting task, not warranted at this early stage of investigation.

Instead, we plot each triaged element (there were 386 in total) for the three tasks in terms of its structural relevance at the instant before triaging the element, and its reuse cost according to the measure in Equation 1. We then roughly overlay the model of Figure 2 on this plot and attempt to fit it according to the data. We try a few variations within the space defined by the degrees of freedom that seem most promising. When some reasonable options are found, a local hill-climbing technique is used in an attempt to improve the results.

1) Estimating model quality: The model quality is evaluated by means of "*ROC analysis*" (e.g., [36]). We can consider rejection recommendations to be positive events, and acceptance recommendations to be negative events (occurrences of no recommendation are ignored in this analysis). A true positive case is one in which a rejection recommendation for an element concurs with an actual rejection of that element during the developer investigation; a false positive is one in which a rejection recommendation for an element coincides with the developer having accepted that element. True and false negatives are analogously defined. (We consider the question of whether the developer's actions represent "reality" in Section V-B.)

The *true positive rate* (TPR) is the ratio of true positive events to the sum of true positive events and false negative events; it represents the rate at which an element that should be rejected is recommended as such. Similarly, the *false positive rate* (FPR) is the ratio of false positive events to the sum of false positive events and true negative events; it represents the rate at which an element that should be accepted is recommended as such.

A tradeoff exists between TPR and FPR; while an ideal recommender will have a TPR of 1 and an FPR of 0, the imprecision inherent in most tasks where recommenders are useful means that this ideal is rarely achievable. A recommender that makes random guesses will describe a straight line in the space spanned by FPR vs. TPR (specifically, the line FPR = TPR); a recommender that is of better quality will tend to lie further away from the line of random guesses [36]. Thus, we can measure the distance between the point plotted and the line of random guesses and use it as a measure of model quality as determined by a particular data set. (ROC analysis usually involves defining curves in the tradeoff space as thresholds are varied, but the high dimensionality of our situation led to our decision to keep this analysis simple.) We can also normalize the measure against the maximum possible distance ($\sqrt{0.5}$) from the line of random guesses so the normalized measure $\hat{D}$ varies in $[0, 1]$ usually (models worse than random are possible, but easily corrected [36]).

2) Model configurations: Our initial models consist of rough attempts at defining half-planes overlain on the data plot. After settling on the most promising candidates, local hill climbing is used to try to improve them relative to $\hat{D}$. Five configurations are tried, as presented in Table II. Configurations 1 and 3 were not competitive and are dropped from further consideration.

Table II
INITIAL CANDIDATE MODEL CONFIGURATIONS.

Cfg	**Regions**	$\hat{\mathbf{D}}$
1	$R_{\text{accept}} = [\text{cost}(x) \leq 1] \vee [\text{cost}(x) \leq 50\,\text{rel}(x) - 30]$, $R_{\text{reject}} = [\text{cost}(x) > 1] \wedge [\text{rel}(x) \neq \text{n/a}] \wedge [\text{rel}(x) > 0.01] \wedge [\text{cost}(x) \geq 50\,\text{rel}(x) + 1]$	0.06
2	$R_{\text{accept}} = [\text{cost}(x) \leq 1] \vee [\text{cost}(x) \leq 50\,\text{rel}(x) - 30]$, $R_{\text{reject}} = [\text{cost}(x) > 1] \wedge \{[\text{rel}(x) = \text{n/a}] \vee [\text{rel}(x) \leq 0.01] \vee [\text{cost}(x) \geq 50\,\text{rel}(x) + 1]\}$	0.32
3	$R_{\text{accept}} = [\text{rel}(x) \neq \text{n/a}] \wedge [\text{rel}(x) > 0.01] \wedge \{[\text{cost}(x) \leq 1] \vee [\text{cost}(x) \leq 50\,\text{rel}(x) - 30]\}$, $R_{\text{reject}} = [\text{rel}(x) \neq \text{n/a}] \wedge [\text{rel}(x) > 0.01] \wedge [\text{cost}(x) \geq 50\,\text{rel}(x) + 1]$	0.12
4	$R_{\text{accept}} = [\text{rel}(x) \neq \text{n/a}] \wedge [\text{rel}(x) > 0.01] \wedge \{[\text{cost}(x) \leq 1] \vee [\text{cost}(x) \leq 50\,\text{rel}(x) - 30]\}$, $R_{\text{reject}} = [\text{rel}(x) = \text{n/a}] \vee [\text{rel}(x) \leq 0.01] \vee [\text{cost}(x) \geq 50\,\text{rel}(x) + 1]$	0.32
5	$R_{\text{accept}} = [\text{cost}(x) \leq 50\,\text{rel}(x) - 30] \vee \{[\text{cost}(x) < 50\,\text{rel}(x) + 1] \wedge [\text{cost}(x) \leq 1]\}$, $R_{\text{reject}} = [\text{cost}(x) \geq 50\,\text{rel}(x) + 1] \vee \{[\text{cost}(x) > 1] \wedge [\text{rel}(x) = \text{n/a}]\}$	0.33

Local hill-climbing succeeded in improving Configurations 2 and 5 to $\hat{D} = 0.37$. For Configuration 2, R_{reject} moved slightly to be bounded by a relevance of 0, and for both configurations, the key bounding line of R_{accept} (with slope and intercept of 50 and –30, respectively) had its slope and intercept changed to 20 and –20, respectively. Configuration 4 was improved marginally (by less than 0.005) by moving the relevance boundary in the same manner (for both R_{accept} and R_{reject}) and by changing the slope and intercept of the key bounding line for R_{accept} to 40 and –20, respectively; the resulting Configuration 4' is

shown in Table III.

Table III
CONFIGURATION 4', SELECTED FOR USE AS THE CALIBRATED MODEL.

$$R_{\text{accept}} = [\text{rel}(x) \neq \text{n/a}] \wedge [\text{rel}(x) > 0] \wedge \{[\text{cost}(x) \leq 1] \vee [\text{cost}(x) \leq 40\,\text{rel}(x) - 20]\},$$
$$R_{\text{reject}} = [\text{rel}(x) = \text{n/a}] \vee [\text{rel}(x) \leq 0] \vee [\text{cost}(x) \geq 50\,\text{rel}(x) + 1]$$

Table IV
CONFUSION MATRIX VALUES FOR THE LEADING CONFIGURATION CONTENDERS.

Cfg	Reject		Accept		None
	TP	FP	TN	FN	
2'	51	31	145	43	116
4'	78	90	128	28	62
5'	51	31	145	43	116

From the confusion matrix values for the calibration tasks (shown in Table IV), we can see that Configurations 2' and 5' yielded identical results for the calibration data. Configuration 4' trades off more false rejects for fewer false accepts and fewer non-recommendations. In our context, false accepts are a potentially more severe problem, since they more likely lead to spurious paths being investigated. (We discuss the implications of false positives in Section VI.) Thus, we favour Configuration 4' (repeated for clarity in Table III and illustrated in Figure 3) and use it as the calibrated model.

B. Retroactive Developer Experiment

We previously performed a controlled laboratory experiment whereby 16 developers (7 industrial; with an overall average of 6 years of development experience) performed two pragmatic reuse tasks, one using Gilligan and the other with standard IDE tools [3]. During the experiment the developers' actions were extensively logged, enabling us to see every decision they made and its timing.

In this evaluation we consider only the 16 experimental trials where developers were using the Gilligan treatment to perform their task—all these were successful by the criteria of the experiment (defined by successfully running a provided test harness). These trials involved two pragmatic reuse tasks. In the first, the developer needed to reuse the QIF parser from the jGnash[5] project (34 kLOC); this involved reusing at least 950 LOC spread between at least 10 classes. In the second, the developer needed to reuse some functionality to retrieve related artists from the aTunes[6] project (33 kLOC); this involved reusing at least 500 LOC spread between at least 10 classes. The jGnash task represented a straightforward reuse task whereby the needed functionality was fairly well modularized; the aTunes task represented a more complex reuse task because the needed functionality was highly coupled to the rest of the project.

For each trial, we utilize only triage actions made by the developer as they created their reuse plan. We consider the set of decisions that remained until the end of the task to have been correct; we consider any decision that was altered to have been incorrect.

We replay each session by executing the decisions in chronological order, recording the values that would have been displayed by Suade's algorithm and our cost measure at the instant the decision was made. Using these values and our calibrated model, we then determine the recommendation that would have been made. From this data we can recreate the developer's environment at each decision point and evaluate if using our combined approach could have been beneficial, by either helping them make their decision more quickly or by avoiding bad decisions. In total, this evaluation considers 2,567 decisions made by the developers.

During their tasks developers made 1,186 accept decisions and 777 reject decisions correctly, by our definition. Table V provides an overview of decisions where developers initially made the wrong choice, along with a measure of how long

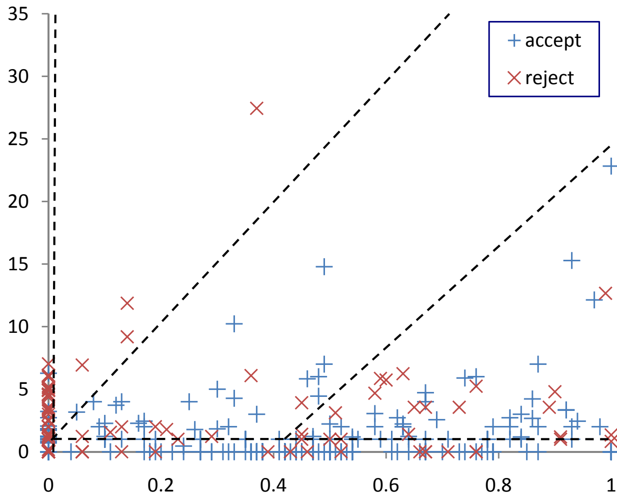

Figure 3. Calibrated recommendation regions in the space of structural relevance versus reuse cost, for Configuration 4'.

[5]http://jgnash.sf.net v1.11.6
[6]http://atunes.org v1.6.0

Table V
INCORRECT DECISIONS (THE DEVELOPER CHANGED THEIR MIND).

Decision Path	Occurrences (#)	Mean Time (seconds)
Accepted ⇒ Accepted	35	596
Accepted ⇒ Rejected	455	107
Rejected ⇒ Rejected	59	387
Rejected ⇒ Accepted	55	178

those incorrect decisions lasted before they were revised; we can clearly see that developers were generally optimistic: the most frequently made mistake (75% of the time) was accepting an element that should have been rejected. Through our observations during the experimental sessions, we can say that these decisions were generally made because the developer failed to appreciate the cost associated with the element they initially thought they should (or should not) reuse.

The results of retroactively generating recommendations for each decision are given in Table VI. Overall, we are able to provide the correct recommendation 64% of the time, no recommendation 11% of the time, and are incorrect 25% of the time. This corresponds well to how often the developers themselves were unable to give the correct decision on their first try (24%; 604 of 2567 decisions).

Examining the participant sessions, we notice that developers sometimes made choices that were questionable; to address these outliers, we look at the decisions that at least 75% of the developers made in common by the conclusion of their task. These results are shown in Table VII. What is surprising here is that they are so similar to Table VI (69% correct, 14% none, 17% incorrect) as we would have expected a larger difference. The greatest change is that our error rate for acceptance recommendations decreases.

Table VI
RECOMMENDER ACCURACY COMPARED TO PARTICIPANT DECISIONS IN EXPERIMENTAL SESSIONS. CONFUSION MATRIX CLASSIFICATIONS ARE SHOWN WHERE APPROPRIATE. PERCENTAGES ARE RELATIVE TO THE CLASS OF DEVELOPER DECISION.

Dev. Decision	Occurrences	Recommendation		
	(#)	Accept (%)	None (%)	Reject (%)
Accept (A)	1186	64 (TN)	7	29 (FP)
Reject (R)	777	18 (FN)	8	75 (TP)
A⇒A	35	80 (TN)	9	11 (FP)
A⇒R	455	29 (FN)	22	50 (TP)
R⇒R	59	37 (FN)	39	24 (TP)
R⇒A	55	58 (TN)	25	16 (FP)

Table VII
RECOMMENDER ACCURACY COMPARED TO DEVELOPER DECISIONS WHERE AT LEAST 6 OUT OF 8 PARTICIPANTS MADE IDENTICAL FINAL DECISIONS.

Dev. Decision	Occurrences	Recommendation		
	(#)	Accept (%)	None (%)	Reject (%)
Accept (A)	952	78 (TN)	11	12 (FP)
Reject (R)	631	18 (FN)	9	74 (TP)
A⇒A	32	91 (TN)	9	0 (FP)
A⇒R	401	29 (FN)	25	46 (TP)
R⇒R	52	40 (FN)	44	15 (TP)
R⇒A	37	57 (TN)	43	0 (FP)

C. Recommendation Case Study

The retroactive experiment provides insight into how well the recommender agreed with the decisions made by developers while they performed their task, but provides little perspective on how these recommendations would actually have influenced how they performed their task. We perform the two experimental tasks with the assistance of the recommender to see how we agree with the recommendations and whether it changes how we approach each task. Both tasks are completed successfully according to the original experiment's criteria.

Figure 4 shows how the recommendations are shown to the user in the modified Gilligan user interface. Recommendations are shown in three colours: red (reject), green (accept), and yellow (no recommendation). We also chose to include the Suade value (Topo column) and cost measure (Cost column) to enable differentiation between multiple elements with the same recommendation.

An overview of the recommendations and corresponding decisions for the two tasks is given in Table VIII. While performing the two tasks we never have to contradict an explicit accept or reject recommendation. When using the recommender we do find the amount of code we reuse increases (by 10% to 15%) as we often chose to reuse "cheap" elements rather than trying to ensure they are absolutely relevant to our task.

The most interesting observation from the case study is the effect the non-recommendations have on how we investigate. While navigating through the system we gravitate to these yellow recommendations and in every case investigate them more carefully than the accept recommendations. By selecting one of these elements, we glance at its dependencies to get a sense of why it has not received an accept or reject rec-

Program Element	Decision	Topo	Cost	Reco
net.sourceforge.atunes.kernel				
Kernel		3.0	3	
DEBUG (Z)		22.0	0	
net.sourceforge.atunes.kernel.modules				
AudioScrobblerArtist		20.0	1.3	
<init>(..)		41.0	0	
getArtist(..)		53.0	7.5	
imageUrl (String)		26.0	0	
match (String)		26.0	0	
name (String)		26.0	0	
url (String)		26.0	0	
AudioScrobblerCache		19.0	6.9	
artistSimilarCacheDir (File)		34.0	0	
getArtistSimilarCacheDir(..)		40.0	1	
getFileNameForArtistSimilar(..)		40.0	1	
getFileNameForArtistSimilarAt(		53.0	4.2	
logger (Logger)		29.0	1	

Figure 4. Gilligan augmented with structural relevance, cost, and recommendation columns.

Table VIII
RECOMMENDATIONS AND CORRESPONDING DECISIONS DURING THE CASE STUDY.

Recommendation	Decision	Events (#)	
		jGnash	aTunes
A	A	72	37
A	R	0	0
NR	A	0	0
NR	R	1	3
R	R	1	5
R	A	0	0
NR⇒A		8	6

ommendation. For example, in the aTunes task, `Closing-Utils.getConnection()` receives a yellow classification; looking at its dependencies we found `Proxy.get-Connection()` that in turn had a yellow classification. By investigating `Proxy` we found it to be dependent on a two `String` fields (`user` and `password`); by accepting these two fields, `Proxy.getConnection()` changed from yellow to green; accepting this caused the initial `getConnection()` method to change to green as well. Essentially, the recommendations shape how we investigate each task and direct us to those areas where we need to most carefully make our decisions. The number of times we investigate an element that is initially yellow but becomes green while exploring its dependencies is listed on the last line of Table VIII.

VI. DISCUSSION

In this section, we examine several issues not discussed to this point, some of which remain to be addressed.

Providing accurate recommendations for complex tasks is hard: Developers differ in their decisions for a variety of reasons including semantics, perception of quality, or personal style. Trying to capture such rationales in an automatic recommender does not seem practicable to us. It is for this reason that our recommender is adaptive to the developer's choices, to actively assist them whatever their rationale for decisions. Looking at the participant session logs, we see a lot of variation in decisions on the same element that did not affect the developers' ability to successfully complete the task. For example, both experimental tasks employ a logging framework to log error messages; many developers decided to reject the logger and all its fields and methods, as it was not directly relevant to their task. Others chose to reuse this functionality because they decided that enhanced logging was worthwhile and was inexpensive to reuse. In these cases the developer must be the final arbiter of what functionality they want to reuse and what they do not want to reuse; a recommender can simply provide them the means to make a more informed decision.

An alternative approach would be to enable developers to provide explicit feedback on recommendations (e.g., through a thumbs up/down feature), allowing the recommendation algorithm to be adapted dynamically, possibly even for all developers within an organization. Such ideas remain to be investigated.

Observations from applying the recommender: Because the participants in the retroactive experiment did not have access to our recommender when they performed their tasks, they were unable to benefit from the properties we observed the recommender to impart in these case studies. From our experience, we believe the recommender significantly enhances a developer's ability to reject an element knowing that it has an unacceptable number of dependencies. At the same time, the recommender also effectively identifies elements that have very low reuse cost, which enables developers to spend less time on these less important elements and instead focus on their task more effectively. From our preliminary case study we believe that developers will agree with the recommendations more often than in the retroactive evaluation, as their presence alters how elements are investigated and reasoned about.

Recommender limitations for our model: Developers investigating pragmatic-reuse plans can accept, reject, or remap elements according to how they want to reuse them. Our recommender does not provide any hints for remapping elements, as it does not leverage any information about the target system. We believe that recommending reasonable remapping opportunities may be possible, but will require significant further research.

Implications of false positives: Ultimately, we expect falsely accepted elements to have little impact on the functionality, but if their numbers grow too large, long-term maintenance costs will be excessive. Thus, while a detailed, manual consideration of all the elements may improve the quality of the reused code, this can be delayed to a later refactoring/clean up stage. In contrast, growing the "no recommendation" region of the model too much would progressively decrease the overall potential for the recommender to produce savings in effort from manual investigation. And finally, false rejection recommendations will truncate useful functionality, so falsely rejected elements are likely to be the most serious case for consideration: while resulting compilation errors are easy to detect, a rejected method call with subtle side effects can be more difficult to notice. However, our sense at this point is that a developer can use the recommendations to focus their attention effectively, thereby saving time, rather than blindly following them. A future study will be needed to test this hypothesis.

Validity: We have compared our recommender against more than 2,500 developer decisions performed by real developers performing two pragmatic reuse tasks in a controlled environment. While this has provided evidence that our recommender can model the kinds of decisions developers really make while planning these tasks, the validity of these observations is hampered by the fact that

the developers did not have the recommender when they performed their tasks. By performing the tasks ourselves we gained initial confidence that, from a usability perspective, the recommender will be valuable for developers. However, ultimately, evidence from the field will provide the best assessment of developers' reactions to the recommendations, and provide the insights we need to improve the technology.

VII. Conclusion

Triaging structural elements while planning pragmatic reuse tasks is a cognitively burdensome, manual process. Developers must constantly flip between different source files to gain a sense of how relevant and expensive an element may be to reuse. While previous work has had success in helping developers to more easily collect and to navigate through this information, we have noticed that developers have a difficult time in identifying and applying notions of reuse cost and structural relevance. Making the right decisions during the triage process is crucial, as a single poor decision may cause an explosion in the scope of the task that makes it infeasible; conversely, rejecting the wrong element may cause the code to not function the way the developer expects.

We have developed a recommendation system intended to facilitate the decision-making process; we have created a ternary recommender that suggests for developers to accept an element that is important, to reject an element that is overly expensive for its apparent relevance, or to more closely investigate its tradeoffs. We evaluated this recommender by comparing it to more than 2,500 decisions made by developers in a previous controlled experiment. Our findings show that our recommender delivers the wrong recommendation less than 25% of the time and is most effective at helping developers identify those dependencies that should be rejected from a task (18% error). Using these recommendations, we expect developers will be able to undertake more complex reuse tasks in less time.

Acknowledgments

We wish to thank our experimental participants, and the anonymous reviewers for their helpful comments. This work was supported by the Natural Sciences and Engineering Research Council of Canada in the form of a Postdoctoral Fellowship and Discovery Grants.

References

[1] R. Holmes and R. J. Walker, "Supporting the investigation and planning of pragmatic reuse tasks," in *Proc. Int'l Conf. Softw. Eng.*, 2007, pp. 447–457.

[2] ——, "Lightweight, semi-automated enactment of pragmatic-reuse plans," in *Proc. Int'l Conf. Softw. Reuse*, 2008, pp. 330–342.

[3] R. Holmes, "Pragmatic software reuse," Ph.D. dissertation, University of Calgary, 2008.

[4] M. P. Robillard, "Topology analysis of software dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 4, pp. 1–36, 2008.

[5] D. Mcilroy, "Mass-produced software components," in *Software Engineering: Report on a Conference by the NATO Science Committee*, 1968, pp. 138–155.

[6] T. A. Standish, "An essay on software reuse," *IEEE Trans. Softw. Eng.*, vol. 10, no. 5, pp. 494–497, 1984.

[7] C. W. Krueger, "Software reuse," *ACM Comput. Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[8] J. S. Poulin, J. M. Caruso, and D. R. Hancock, "The business case for software reuse," *IBM Syst. J.*, vol. 32, no. 4, pp. 567–594, 1993.

[9] B. Boehm, "Managing software productivity and reuse," *Computer*, vol. 32, no. 9, pp. 111–113, 1999.

[10] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Commun. ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[11] G. Succi, L. Benedicenti, and T. Vernazza, "Analysis of the effects of software reuse on customer satisfaction in an RPG environment," *IEEE Trans. Softw. Eng.*, vol. 27, no. 5, pp. 473–479, 2001.

[12] W. B. Frakes and G. Succi, "An industrial study of reuse, quality, and productivity," *J. Softw. Syst.*, vol. 57, no. 2, pp. 99–106, 2001.

[13] S. A. Ajila and D. Wu, "Empirical study of the effects of open source adoption on software development economics," *J. Softw. Syst.*, vol. 80, no. 9, pp. 1517–1529, 2007.

[14] R. Prieto-Díaz, "Status report: Software reusability," *IEEE Softw.*, vol. 10, no. 3, pp. 61–66, 1993.

[15] M. B. Rosson and J. M. Carroll, "The reuse of uses in Smalltalk programming," *ACM Trans. Comput.-Hum. Interact.*, vol. 3, no. 3, pp. 219–253, 1996.

[16] B. M. Lange and T. G. Moher, "Some strategies of reuse in an object-oriented programming environment," in *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, 1989, pp. 69–73.

[17] J. R. Cordy, "Comprehending reality: Practical barriers to industrial adoption of software maintenance automation," in *Proc. IEEE Int'l Wkshp. Progr. Comprehension*, 2003, pp. 196–205.

[18] M. Toomim, A. Begel, and S. L. Graham, "Managing duplicated code with linked editing," in *Proc. IEEE Symp. Visual Lang. Human Centric Comput.*, 2004, pp. 173–180.

[19] C. Kapser and M. W. Godfrey, "'Cloning considered harmful' considered harmful: Patterns of cloning in software," *Empir. Softw. Eng.*, vol. 13, no. 6, pp. 645–692, 2008.

[20] R. W. Selby, "Enabling reuse-based software development of large-scale systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 495–510, 2005.

[21] G. Fischer, "Cognitive view of reuse and redesign," *IEEE Softw.*, vol. 4, no. 4, pp. 60–72, 1987.

[22] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 529–536, 2005.

[23] A. Sen, "The role of opportunism in the software design reuse process," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 418–436, 1997.

[24] M. Morisio, M. Ezran, and C. Tully, "Success and failure factors in software reuse," *IEEE Trans. Softw. Eng.*, vol. 28, no. 4, pp. 340–357, 2002.

[25] T. Ravichandran and M. A. Rothenberger, "Software reuse strategies and component markets," *Commun. ACM*, vol. 46, no. 8, pp. 109–114, 2003.

[26] G. Caldiera and V. R. Basili, "Identifying and qualifying reusable software components," *Computer*, vol. 24, no. 2, pp. 61–70, 1991.

[27] F. Lanubile and G. Visaggio, "Extracting reusable functions by flow graph-based program slicing," *IEEE Trans. Softw. Eng.*, vol. 23, no. 4, pp. 246–259, 1997.

[28] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 213–225, 2005.

[29] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Softw.*, vol. 12, no. 6, pp. 17–26, 1995.

[30] O. A. L. Lemos, S. K. Bajracharya, and J. Ossher, "CodeGenie: A tool for test-driven source code search," in *Proc. ACM Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, 2007, pp. 917–918.

[31] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, p. 1–38, 2007.

[32] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with Dora to expedite software maintenance," in *Proc. IEEE/ACM Int'l Conf. Automated Softw. Eng.*, 2007, pp. 14–23.

[33] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending random walks," in *Joint Proc. Europ. Softw. Eng. Conf./ACM SIGSOFT Int'l Symp. Foundations Softw. Eng.*, 2007, pp. 15–24.

[34] F. Weigand-Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in *Proc. Int'l Conf. Softw. Eng.*, 2007, pp. 780–783.

[35] R. Holmes and R. J. Walker, "Task-specific source code dependency investigation," in *Proc. IEEE Int'l Wkshp. Visual. Softw. Underst. Analys.*, 2007, pp. 100–107.

[36] T. Fawcett, "An introduction to ROC analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, 2006.