An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks

SCOTT D. FLEMING, Oregon State University and University of Memphis
CHRIS SCAFFIDI, Oregon State University
DAVID PIORKOWSKI, Oregon State University
MARGARET BURNETT, Oregon State University
RACHEL BELLAMY, IBM Research
JOSEPH LAWRANCE, Massachusetts Institute of Technology
IRWIN KWAN, Oregon State University

Theories of human behavior are an important but largely untapped resource for software engineering research. They facilitate understanding of human developers' needs and activities, and thus can serve as a valuable resource to researchers designing software engineering tools. Furthermore, theories abstract beyond specific methods and tools to fundamental principles that can be applied to new situations. Toward filling this gap, we investigate the applicability and utility of Information Foraging Theory (IFT) for understanding information-intensive software engineering tasks, drawing upon literature in three areas: debugging, refactoring, and reuse. In particular, we focus on software engineering tools that aim to support information-intensive activities, that is, activities in which developers spend time seeking information. Regarding applicability, we consider whether and how the mathematical equations within IFT can be used to *explain why* certain existing tools have proven empirically successful at helping software engineers. Regarding utility, we applied an IFT perspective to identify recurring *design patterns* in these successful tools, and consider what *opportunities for future research* are revealed by our IFT perspective.

Categories and Subject Descriptors: [**D.2.6 Programming Environments]**: Integrated environments, Interactive environments, Programmer workbench

General Terms: Design, Human Factors

Additional Key Words and Phrases: Information foraging, software maintenance

1. INTRODUCTION

Most software engineering (SE) research aims to help human software developers with their work, but to date, SE research has made little use of theories of human behavior. This is unfortunate, because it denies these works the benefits that such theories can bring. The essence of theories is abstraction—from instances of new approaches to concepts and principles. In the realm of human behavior, these abstractions can then produce explanations of *why* software engineering tools succeed at supporting the efforts of software developers (or why some tools that were expected to succeed did not), so that a principled understanding can emerge from prior researchers' efforts.

A large group of software engineering researchers known as the SEMAT initiative[1] recently discussed this point, proposing that a stronger grounding in solid theoretical foundations is needed overall in software engineering research, including the aspects that they term *people issues*. SEMAT's concerns dovetail with the findings of a recent systematic review of empirical software engineering research, which showed that despite considerable research on the psychology of programming, there have been only isolated successes in the application of human behavioral theories to software engineering [Hannay et al. 2007]. In particular, only 24 out of the 100 empirical software engineering papers considered by that review used theory at all, and in those papers, only 10 out of the 40 theories used were psychological theories. SEMAT identified several consequences of theory's underdeveloped role in software engineering research. For instance, a scientific consequence is the

[1] http://www.semat.org/

proliferation of tools and methods "with differences little-understood and artificially magnified." A practical consequence is limited impact of academic research on industry practice.

One theory that has been practically useful in design is Information Foraging Theory (IFT). The theory explains and predicts how people navigate in response to the information in their environment. It has proven particularly helpful in explaining what makes for an effective Web design (i.e., such that users can find their desired content easily and efficiently) and has a track record of successfully predicting how people will seek information on the Web (e.g., [Pirolli and Card 1999; Chi et al. 2001]). Building on this success, researchers have used IFT to design effective tools to help creators and users of Web sites (e.g., [Aula et al. 2005; Chi et al. 2003; Wexelblat and Maes 1999]). In response to the empirical evidence of both IFT's validity in behavior prediction and its practical utility for Web site and tool builders, IFT has become accepted as a valid and useful theory among researchers in HCI and psychology [Olson and Olson 2003].

This paper explores whether and how IFT can be harnessed to aid researchers and tool builders in the design of software engineering tools. IFT has several attributes that suggest that it is well suited to software engineering. First, IFT addresses the pervasive software engineering problem of *information seeking,* which accounts for over one third of the time that software developers spend during maintenance activities [Ko et al. 2006]. Second, IFT is *parsimonious*: It uses a small set of constructs to describe and make predictions about a wide variety of information-seeking phenomena. Third, IFT is *predictive*, meaning that it can be and has been experimentally operationalized and tested both by the theory's initial designers [Pirolli and Card 1999; Pirolli and Fu 2003; Pirolli and Card 1998] and by numerous other scientists working in the domains of user-web interaction [Wexelblat and Maes 1999; Vigo et al. 2009; Budiu et al. 2009; Aula et al. 2005] and, more recently, software engineering [Lawrance et al. 2008a; Lawrance et al. 2011; Lawrance et al. 2010].

Therefore, in this paper we investigate (1) the applicability and (2) the utility of IFT for SE tool design. We do so by applying an IFT perspective to tools from three areas of the SE literature: debugging, refactoring and reuse. We chose three areas in order to consider IFT's ability to unify disparate SE activities under a common abstraction. We focus on tools from these areas that aim to support information-intensive activities—that is, activities in which developers spend time seeking information. To investigate IFT's applicability, we consider *whether and how* IFT's mathematical equations can be used to explain why certain existing tools have proven empirically successful at helping software engineers. To investigate IFT's utility, we distill from the tools *design patterns* that can be used to guide the design of future SE tools. We also consider *opportunities* for future research revealed by viewing these SE tools through the lens of IFT.

The remainder of this paper is organized as follows. Section 2 reviews IFT's key mathematical equations and empirical success as a predictive theory outside of software engineering. Section 3 draws upon existing literature to show the applicability of IFT to software engineering tasks that involve information-seeking by software developers. Section 4 demonstrates the utility of IFT by using an IFT perspective to identify twelve design patterns that were instantiated in multiple areas. Section 5 builds on Section 4's utility theme by discussing new opportunities for future research revealed by our IFT perspective. Section 6 summarizes related work within software engineering research. Section 7 concludes the paper with a summary of our key results.

2. BACKGROUND: INFORMATION FORAGING THEORY

2.1 Roles for Information Foraging Theory

Information Foraging Theory's originator, Peter Pirolli, suggests two roles for IFT: "to *explain and predict* (emphasis added) how people will best shape themselves for their information environments and how information environments can best be shaped for people" [Pirolli 2007]. However, Sjoberg et al. point out that a theory can play any or all of *four* roles [Sjoberg et al. 2008], and IFT has been used for all four. One role is to describe or analyze observed phenomena, helping people to understand "what is." For example, Ko et al. used IFT to interpret results from an empirically based model of program comprehension [Ko et al. 2006]. A second role is to explain observed phenomena, helping people to understand "why" events occur. A third is to predict phenomena by forecasting future events (though not necessarily explaining reasons why). Pirolli and colleagues used IFT both to explain why people choose to click particular links on a Web page and to predict which links people will click (e.g., [Chi et al. 2001]). A fourth role is to guide design and action, describing "how to do" something, such as how to design an effective SE tool. For example, researchers of Web behaviors have exploited the theory's predictive power to inform the design of Web sites, thus improving the usability of such sites (e.g., [Chi et al. 2003]).

2.2 IFT constructs and propositions

In IFT, a person, the *predator*, seeks information within an information environment. The environment has a *topology* made up of *information patches* (e.g., documents) connected by traversable *links* (e.g., clickable hyperlinks, menus, scrolling, etc.). Each link has a cost of traversal (e.g., time to get from one end to the other, where the time is influenced by system performance and the human's cognitive and physical speed). At any moment, the predator is within a particular patch and can navigate to another patch by traversing an outgoing link. Figure 1 graphically depicts an example topology.

Each patch contains *information features* (e.g., words, phrases, diagrams, etc.) that the predator can process. The predator seeks to find and process a particular set of information features, the predator's *information goal*. The information features that are elements of the information goal set are termed *prey*. Similar to links, each information feature has a cost (e.g., time to read and understand). Some features, known as *cues*, are associated with outgoing links. Cues provide the predator with hints about what distal information features a link leads to. For example, cues might be operationalized as the words in a hyperlink's human-readable text. Figure 2 zooms in on the example topology from Figure 1 to reveal the contents of two patches.

In searching for prey, the predator is continually faced with three choices. One choice is to process information features in the current patch. Another choice is to move to an adjacent patch via an outgoing link. A final choice is to change the environment—an option known as *enrichment*. As an example of enrichment, the predator could add a new patch to the topology (e.g., a list of electronic notes) or create a new link (e.g., a bookmark). However, in operationalizations of the theory, real-world environments often constrain the predator's enrichment options. For example, consider a person seeking information on the Web. The person (predator) can perform enrichment by using a search engine, such as Google, to generate a patch; however, the person cannot enrich the environment by arbitrarily modifying the hyperlinks on a Web page because the Web browser interface does not allow it.

The central prediction of IFT is that the predator makes choices that attempt to maximize valuable information gained per cost of interaction, an intent characterized by the equation:

$$\text{Predator's desired choice} = \max\left[\frac{V}{C}\right] \tag{1}$$

where V is value of information gained and C is cost of interaction, which includes the cost of enriching the environment, the cost of traversing links, and the cost of processing information features. In general, predators' attempts to maximize this ratio are hampered because they do not know which choices will achieve the maximum.

Given this limitation, IFT predicts that predators will make choices based on their estimation of cost and value. That is, they tend to maximize *expected* value per *expected* cost, a tendency characterized by the equation:

$$\text{Predator's selected choice} = \max\left[\frac{\mathrm{E}(V)}{\mathrm{E}(C)}\right] \tag{2}$$

where E(V) and E(C) are, respectively, value and cost expected by the predator given available information. How close the predator comes to actually maximizing value per cost depends on the accuracy of the predator's expectations.

When leaving a patch, the predator relies on the local cues to decide which link to traverse. In particular, the predator uses these cues to assess the value to be gained and the cost to be incurred by following a given link. Following from Equation 2, the predator will favor links whose cues suggest that the links lead to valuable prey. Formally, suppose that the predator seeks a set of prey G and has a set of outgoing links L to choose from. Each link $l \in L$ has a set of associated cues J_l (e.g., words in a hyperlink). IFT predicts that predators will tend to choose link l that maps to the greatest value in the following function IS:[2]

$$IS(l) = \sum_{i \in G}\left(\sum_{j \in J_l} W_j S_{ji}\right) \tag{3}$$

[2] Pirolli [2007] includes an additional term B_i, which is only meaningful in the context of multiple goals. We omit the term here for clarity.

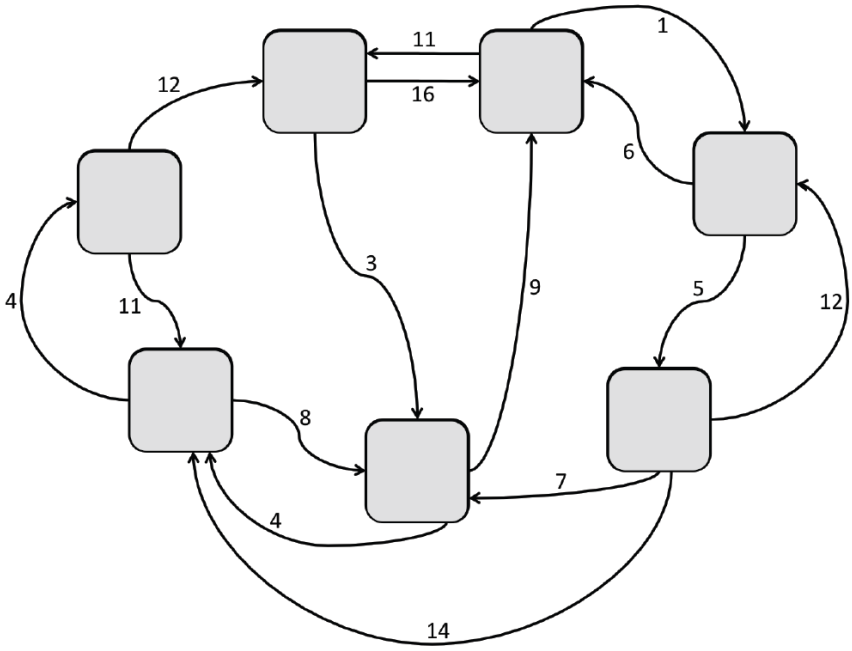


Fig. 1. Example topology. Each rounded square is an information patch. Each directed edge is a navigable link from one patch to another, and the weight on each link indicates the cost of traversing the link. (Figure adapted from [Pirolli 2007, Fig. 7.1].)

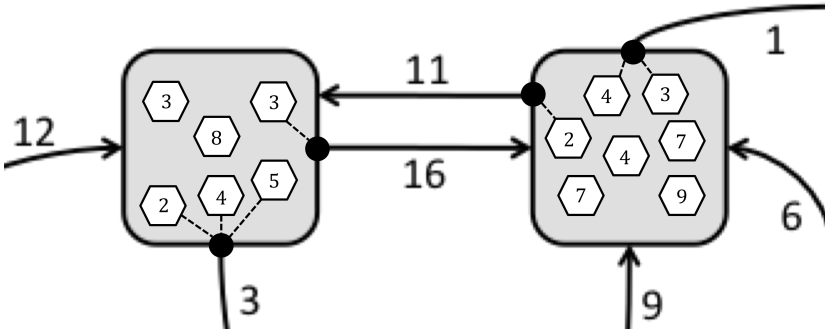


Fig. 2. Example of two information patches. Each patch contains a set of information features, depicted as hexagons. Some information features are cues, which are associated with outgoing links; dashed lines denote these associations. Each hexagon is also annotated with a processing cost (shown inside the hexagon). (Figure adapted from [Pirolli 2007, Fig. 4.8].)

Information scent S_{ji} is the predator's estimation of the likelihood of some prey $i \in G$ being at the other end of the link in the presence of some cue $j \in J_l$. Scent is represented as a nonnegative number in the equation. For example, one common operationalization of S_{ji} is based on measures of textual similarity (although other operationalizations are possible). The predator's perception of scent based on a particular cue j is influenced by the amount of attention W_j (a nonnegative number) that the predator pays to j. For example, researchers have operationalized W_j as a monotonically increasing function of the cue's visual size [Olston and Chi 2003]. This operationalization captures the idea that predators give less attention to visually small cues, and thus, perceive less information scent based on such cues. The position of links and their associated cues within a patch can influence which link a predator chooses to traverse [Fu and Pirolli 2007]. Because there is a cost associated with

assessing each link (i.e., the cost of processing the cues), the predator may not assess all the links in a patch. Instead, the predator may assess only some of the links, eventually settling upon a link that is "good enough"—a process known as *satisficing*. For instance, in one operationalization of the theory, predators assessed links from top to bottom and left to right on a page and chose links based on a combination of *IS* and how many links they had assessed so far. In an experiment, this operationalization predicted the navigation choices of Web users more accurately that an operationalization that assumed a global assessment of links [Fu and Pirolli 2007].

Table 1 summarizes the constructs of IFT.

2.3 Corollary predictions and empirical validation in the domain of user-Web interaction

The literature contains a considerable body of work validating the predictive power of IFT for the domain of user-Web interaction. An early empirical study [Pirolli and Card 1999] investigated users (predators) foraging for information in a Web-like document-browsing system. The study operationalized cost C as the time a user spent navigating among and reading documents, and value V as expert assessment of whether or not a document was relevant to the user's task. The study found that the users followed hyperlinks (links) that appeared to have the highest expected value for a given cost, as predicted by IFT. A subsequent study of Web users replicated this result, showing that users tended to leave Web sites when the expected value-to-cost ratio elsewhere exceeded the expected value-to-cost ratio of staying at the same site, as predicted by IFT [Pirolli and Fu 2003].

Table 1. IFT's key constructs.

Construct	Description
Topology	Collection of information patches and links between those patches within a particular information environment
Information patch	Region in the topology that contain information features
Links L	Traversable arcs between patches
Information features	Elements of the environment that the predator can process to gain knowledge
Cues J_l	Set of information features associated with a particular link $l \in L$
Predator	Person in search of information
Information goal G	Set of information features that the predator wants to find
Prey $i \in G$	Individual features in the goal set G
Information scent S_{ji}	Given a link with associated cue j, the predator's estimation of the probability that traversing the link will lead to prey i
Attention W_j	Amount of attention that a predator pays to a particular cue j
Information value V	Benefit of processed information to the predator
Interaction cost C	Costs incurred by the predator during foraging, including the cost of enriching the environment, the cost of traversing links, and the cost of processing information features
Expected value E(V)	Value that the predator anticipates gaining through a particular course of action (e.g., following a particular link)
Expected cost E(C)	Cost that the predator anticipates incurring in following of a particular course of action

Additionally, researchers have empirically validated several corollary predictions of IFT. Corollary predictions relate to how people's experiences are affected by phenomena captured in specific terms of IFT's equations. One such corollary prediction is that if the costs of certain navigations are lowered (reducing the denominator C in Equation 1), then users are more likely to select those navigations. Researchers tested this prediction in two studies [Pirolli and Card 1998]. One study compared a standard Web browser to an experimental browser that used prefetching to reduce the cost of navigating between pages. Another study compared a standard browser to one that inserted "shortcut" hyperlinks between certain pages. As predicted, participants in both studies favored the lower cost links.

Another corollary prediction is that the more accurately cues reflect the value and cost of prey (i.e., improving the accuracy of S_{ji}), the more effectively people will use those cues to find their prey. For example, one empirical study demonstrated that users tend to complete foraging tasks more quickly when hyperlinks are annotated with a count of how many people (total) have ever clicked on that link, which implicitly indicates whether a given link points to a page of generally high value [Wexelblat and Maes 1999]. In another test, Vigo et al. predicted and empirically confirmed that blind users would be able to more quickly find information if hyperlinks are annotated with a measure of whether the target pages contain features accessible by blind users, thus helping to align such users' expectations about the cost of processing a target page with the actual cost (i.e., helping to align E(C) with C) [Vigo et al. 2009]. A third prediction is that novice Web users working on novel tasks have the highest probability of choosing links ranked with the greatest textual similarity to the users' stated goal [Pirolli and Fu 2003]. The rationale is that if a person with little knowledge of a domain is asked to look for the answer to a

specific question, then the similarity of the words in the question to hyperlink text is the main form of scent available. Again, experiments confirmed the prediction [Pirolli and Fu 2003].

Yet another corollary prediction is that if users can reshape the environment through enrichment activities, then they will choose to do so in a way that enables them to find prey more efficiently. For example, if people can leave tagged bookmarks in the environment, then they will be able to re-find information more quickly. Moreover, if the cost of this enrichment action is lowered, then people perform the action more [Budiu et al. 2009]. These predictions were empirically confirmed [Budiu et al. 2009]. As another example, if users can create a page of search results and subsequently refine or filter those results, then it was predicted and confirmed that they will be able to find prey more quickly overall [Pirolli and Card 1995]. As a third example related to enrichment activities, it has been predicted and confirmed that more experienced foragers would make better enrichment choices than less experienced users, because experience should provide improved estimates of the value and cost of those choices [Aula et al. 2005].

The common theme of these corollary publications is the explicit use of IFT by researchers to generate and test concrete predictions of user behavior in specific situations on the Web. This predictive power contributes to IFT's utility to tool designers, as has already been demonstrated in the domain of Web information-seeking, and which we next explore bottom-up in the domain of software engineering.

3. USING IFT TO UNDERSTAND THREE AREAS OF SOFTWARE ENGINEERING

To assess the applicability and utility of IFT to software engineering, each subsection below draws upon literature in one of three areas: debugging, refactoring, and reuse. We selected these areas because they provide a range of information-intensive activities that are supported, at least in part, by a variety of SE approaches that have been instantiated as tools. These selection criteria provide the breadth needed to consider whether (and how) abstract IFT constructs can *describe* software engineering activities across areas (and to identify activities that do not correspond well to IFT constructs). The criteria also provide the variety needed to consider whether IFT constructs can provide a unifying conceptual framework that can *explain why* a *cross-section* of SE tools have proven successful.

3.1 Debugging

Debugging can be highly information-intensive, requiring developers to gather information for generating hypotheses about what is wrong, to gather information for confirming or refuting (and refining) hypotheses, and to gather information for determining how to fix the bug [Brooks 1983]. In IFT terms, the predator is the developer, whose prey can potentially include chunks of source code, documentation, e-mails, bug reports, and other documents [Cubranic et al. 2005]. These artifacts form the patches in a topology whose navigation links are typically provided by a development environment. For example, a developer might click on a class name to navigate to that class's source in Eclipse[3] or to a UML diagram of the class in Rational Rose[4]. With the IDEs used in industrial practice, it is much easier to navigate between certain patches (particularly code files) and relatively hard to navigate between others (e.g., from code to e-mails). These differences give rise to patches that are not always structured in a way that facilitates debugging. One way

[3] http://www.eclipse.org/

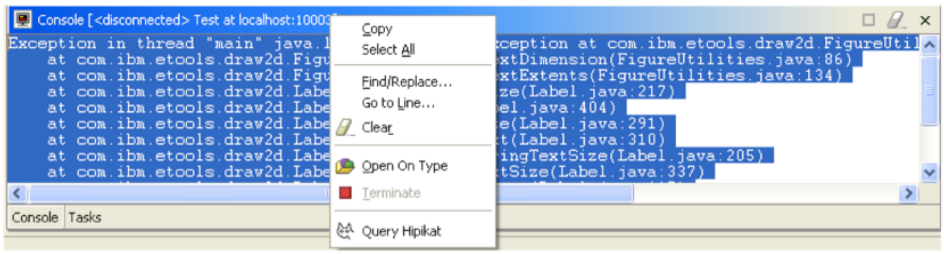
[4] http://www.ibm.com/software/rational/

that a tool can facilitate debugging is by helping developers as they enrich this topology (e.g., Hipikat, below).

Without environment support, foraging during debugging may be tedious and costly (i.e., high *C*). For example, the foraging efforts of a developer debugging with only a basic text editor may involve inserting diagnostic print statements, viewing the output, navigating via the operating system from the output screen to the code, scrolling through the code, navigating to other files (again via the operating system), and so on. In contrast, most modern debuggers allow developers to set breakpoints that link directly from an executing program to code, which can considerably reduce the cost of navigating from an executing program's interface to its code (and thus can reduce *C*). In IFT terms, setting breakpoints enriches the environment by creating low-cost links. Other tools from the literature go even further in helping developers to enrich the topology with links that reduce navigation cost and with cues that guide the predator to valuable prey (Table 2).

For example, Hipikat supports foraging by reducing the cost of navigation among code and non-code artifacts, such as bug reports, CVS logs, and e-mails (and thus reducing *C* overall) [Cubranic et al. 2005]. Non-code artifacts document past experiences of developers working on the project, so such artifacts often provide information about how to fix bugs. For example, e-mails might explain subtle interactions between pieces of code. Hipikat enables the developer to enrich the topology by specifying a query consisting of one artifact (e.g., thrown-exception data; illustrated in Figure 3a) or a set of keywords (e.g., words in an exception stack trace). Hipikat attempts to infer the developer's goal *G* by identifying artifacts that are lexically similar to the query. It creates a patch of links to the identified artifacts (Figure 3b). Each link is labeled with cues indicating a rationale for why the link was created as well as an estimate of how useful the link is for obtaining useful information. These cues aim to engender accurate scent S_{ji} in the mind of the developer.

(a) The developer selects the output of a thrown exception and queries Hipikat.



(b) Based on the query, Hipikat creates a patch of links to relevant artifacts. Each search result has associated cues that indicate rationale and confidence.

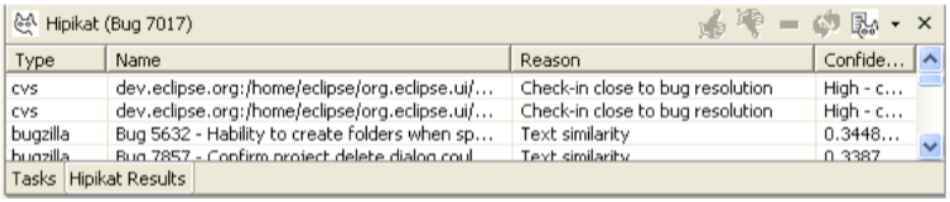


Fig. 3. The Hipikat Eclipse plug-in. (Images courtesy of Davor Cubranic)

Other tools seek to facilitate foraging by reducing the cost of navigation (and thus *C* overall) by enabling developers to quickly "ask questions". For instance, in Whyline [Ko 2008] the developer produces a failing run of the program that is used to generate a list of links to methods executed during that failed run. For example, the developer can execute a program that draws an incorrectly colored line on-screen, and then click to retrieve links to the methods that were called during and leading up to the line's creation (Figure 4a). This feature of Whyline makes it inexpensive to navigate from an observable output to the line(s) of code that produced that output. These lines constitute Whyline's approximation of the developer's goal *G*. For example, the developer could navigate to code answering the question, "Why did color=rgb(0,0,0)?"—thereby selecting a link to traverse based on semantics rather than just lexical similarity (Figure 4b). From an abstract IFT perspective, these cues aim to engender highly accurate scent S_{ji} in the minds of predators. Similar concepts, like Ferret [De Alwis and Murphy 2008] and Information Fragments [Fritz and Murphy 2010] make it easier for the programmer to ask and answer particular questions by using a context-sensitive view that identifies relationships between different artifacts as well as people in a project. In particular, information fragments can display links to artifacts in response to a "What code files is someone working on?" question and thus reducing cost *C*.

Still other tools, such as Mylyn (formerly Mylar) [Kersten and Murphy 2006], TagSEA [Storey et al. 2007], and Team Tracks [DeLine et al. 2005], reduce the cost *C* of foraging by enabling developers to find and organize artifacts based on prior access patterns. That is, these tools are based on the assumption that the goals *G* of developers tend to include artifacts that prior developers working on a similar problem found to be related. In IFT terms, these tools enable developers to enrich the topology by generating patches corresponding to tasks or concerns. For example, when the developer specifies that a certain task has begun, Mylyn logs the code visited while the developer performs the task. If that developer or another person needs to revisit the task, Mylyn provides a list of links to previously visited code. With TagSEA, developers explicitly assign tags to regions of code; developers can then filter code through a tag, share tag sets, and navigate back to tagged code. Team Tracks automatically infers relatedness of artifacts based on how often developers navigate between artifacts. Team Tracks is based on the assumption that methods

Table 2. Examples of tools that facilitate debugging

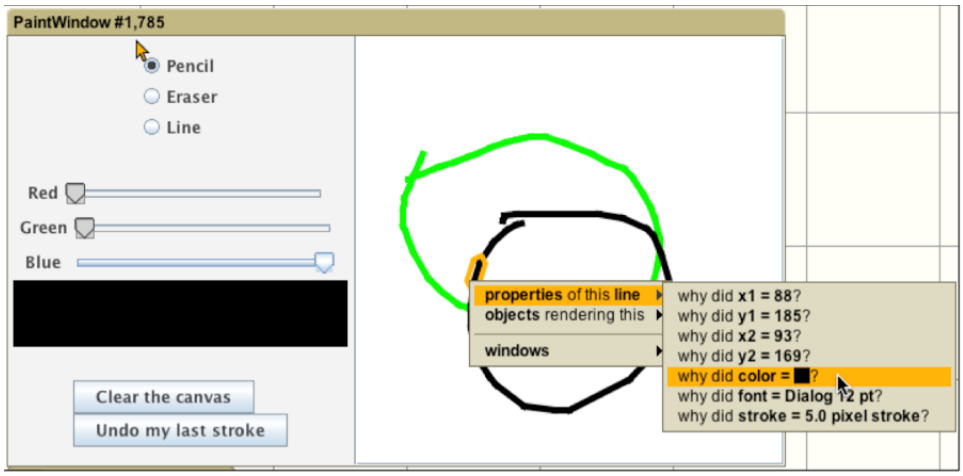
Tool	Topology enhancement	Roles of cues	Reference
Hipikat	Creates links from search results to artifacts	Uses lexical similarity of textual cues to identify search results; labels each search result with a cue that gives the rationale for that result	[Cubranic et al. 2005]
WhyLine	Creates links from program output to source code	Uses temporal sequencing of events to generate links; labels each new link with a cue in the form of a semantically meaningful question	[Ko 2008]
Mylyn	Uses task context to identify code artifacts; Filters links between code artifacts for selected task context	Filters cues in tree and list views for selected task context	[Kersten and Murphy 2006]
Team Tracks	Creates links between code artifacts	Uses developers' prior navigation actions to identify artifacts; annotates each new link with cues in the form of arrows and stars indicating why the link was provided	[DeLine et al. 2005]

accessed in close succession are related and provides a query feature that, given a particular method, returns related methods. To help developers assess the value of links, Team Tracks annotates links with iconic cues indicating why it generated each link.

Although IFT was not explicitly used to design any of these tools, it unifies their commonalities through a single explanation for their empirical success: IFT predicts that reducing the cost C by reducing the cost of navigating to valuable prey (e.g., through enrichment) will increase the benefit-to-cost ratio of the optimal choice (Equation 1), and also will increase the predator's attraction to the most valuable prey (Equation 3), thereby helping to align the predator's expectations of value and cost (Equation 2) with reality and thus improving the chances that the predator will make the optimal choice.

The results of empirical evaluations of these tools are consistent with this explanation. For example, one study found that Hipikat helped newcomers to a project discover information about subtle "special cases," enabling these newcomers to complete tasks almost as correctly as developers more familiar with the project [Cubranic et al. 2005]. Another study showed that developers who used Whyline had a higher rate of success on debugging tasks and finished tasks twice as quickly as developers who used traditional debugging tools. Studies also showed that Mylyn and Team Tracks enabled developers to more effectively focus on code that needed editing [DeLine et al. 2005; Kersten and Murphy 2006].

(a) Whyline has recorded a run of a buggy painting program. The black line should be blue. Clicking it causes Whyline to produce a patch of hyperlinks to locations in the code that are related to the line. Each link is labeled with a cue in the form of a "why" question that the associated code answers.



(b) By clicking the "why did color =...?" question above, the developer traverses the link to the location in the source code where the color was set. Clicking elements in the code, such as the variable `color`, produces more patches of links labeled with questions.

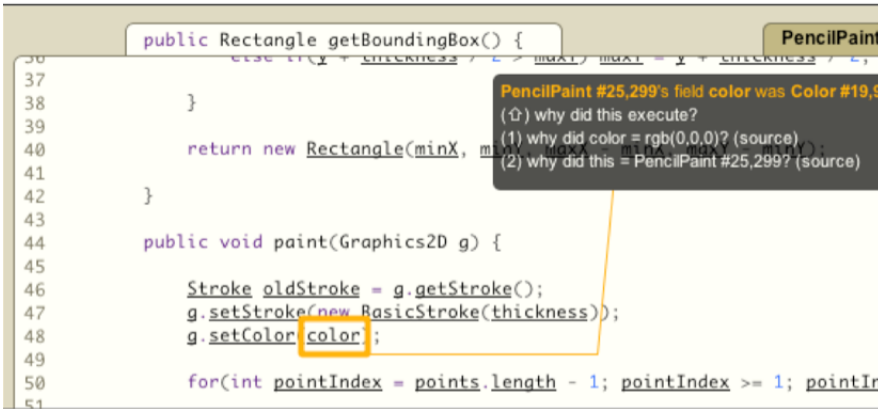


Fig. 4. The Whyline tool [Ko 2008]. (Images courtesy of Andrew Ko)

Thus, IFT allows us to abstract above specific tools, by providing theoretical constructs that capture the shared essence of these debugging tools: They all reduce developers' navigation effort, and thereby increase the ratio in Equation 1 through generation of patches (e.g., through automated search features), through other enrichment of topology (e.g., through breakpoints), and through enhancement of cues (e.g., through manual tagging). In the next section, we consider whether we can relate refactoring tools' to the same abstract constructs in IFT.

3.2 Refactoring

Refactoring is the transformation of code to improve maintainability without changing functionality [Fowler and Beck 1999]. In refactoring, first, the developer identifies problems that call for refactoring. Second, having identified a problem, the developer selects a refactoring to fix it. Third, the developer locates all the code to modify for the refactoring and applies the changes. The first and third activities in the process are foraging oriented, so we discuss them in Sections 3.2.1 and 3.2.2, respectively; see Table 3 for examples of IFT-relevant tool support for these activities. The second activity is not a foraging activity and thus is not well-suited to IFT constructs.

3.2.1 Information foraging when identifying problems that call for refactoring

From an IFT perspective, the predator is a developer seeking prey that contains information about opportunities to refactor. These prey primarily comprise application code (which will be refactored), although test code and documentation can also provide information about some problems (as discussed below). As in debugging, the task environment can be operationalized as a patchy topology consisting of these artifacts, some connected via navigable links. However, the cues and scent involved in refactoring differ from those in debugging.

Specifically, developers look for code with "bad smells" [Fowler and Beck 1999]. Such smells are code structures that suggest the possibility of refactoring because they tend to inhibit maintenance. A commonly mentioned example is code clones, which are considered a bad smell because they typically must be kept in sync during maintenance [Geiger et al. 2006]. This is not to say that clones always contain bugs. Indeed, empirical studies suggest that clones are generally less likely to contain bugs than other code [Rahman et al. 2010]. However, refactoring of clones is sometimes desirable to reduce the need for parallel edits during maintenance. Other bad smells include long parameter lists in methods, unusually long classes, and *speculative generality*—code that is never called (except by test code) or that is otherwise superfluous [Fowler and Beck 1999].

Despite the similarity between the words *smell* and *scent*, smells are more like cues in IFT than scent. Smells are human-perceptible phenomena (bad code structures) suggesting that code will be difficult to maintain. The smells act as signposts that engender scent in the mind of the developer and that the developer may follow to code in need of refactoring.

Without environment support, finding certain bad smells can be time-consuming (high C). For example, developers could find duplicate code by manually navigating through files and trying to remember which lines have been seen before. The predominant approach for accelerating this process (i.e., reducing C) is to automatically scan code for bad smells. In IFT terms, this approach approximates the developer's goal G as bad smelling code. The developer can use this feature to enrich the environment by creating a patch of scan results. This enrichment collects in one place prey (or links to prey) that might otherwise have required extensive foraging to detect.

Different search-based tools rely on different goal approximations (Table 3). For example, Crocodile uses metrics to find excessively complex code [Simon et al. 2001]. In contrast, Soul finds speculative generality by identifying application code that is only called from test code [Mens et al. 2003]. It uses similar structural rules to find additional smells, as does jCOSMO [van Emden and Moonen 2002]. Another tool, Duploc, locates code clones by computing similarity between classes [Ducasse et al. 1999]. Because goal approximation can be inaccurate, many tools enable users to refine search results (e.g., jCOSMO's "pruning" feature for filtering [van Emden and Moonen 2002]).

These search-based tools commonly use their goal approximations to create a patch in the form of a list of search results. However, some tools take the output of smell detectors as inputs to provide additional functionality. For example, CloneTracker takes a set of clones and generates an abstract representation of them, which enables the tool to keep track of clones' locations even as code evolves [Duala-Ekoko and Robillard 2007]. That way, if a developer later refactors or otherwise transforms one clone, the tool can quickly find the other clones that need to be transformed—and in fact, this powerful tool can automatically propagate many such transformations to the other clones. Codelink is a similar tool that helps developers to find, manage, and perform simultaneous edits to multiple clones [Toomim et al. 2004].

IFT predicts that reducing cost C by lowering the cost of navigating to valuable prey, as these search-based tools do, will increase the ratio in Equation 1, thus enabling developers to acquire prey more efficiently and to discover prey that might have otherwise been missed. Studies confirm this prediction for the above search-based tools. Soul identified 65 opportunities for refactoring in three code bases. These results reduced the cost of foraging for 60 refactorings to 0 (5 of the suggested refactorings were false positives, where smells did not really need refactoring); Soul even found useful opportunities to refactor a code base considered very stable [Mens et al. 2003]. In another study, six clone-detection algorithms were able to analyze code bases up to 235 KLOC in just minutes, vastly less than the cost C of a human searching by hand [Bellon et al. 2007]. IFT allows us to generalize the results of these studies in terms of Equation 1: When searching for refactoring opportunities,

Table 3. Examples of tools that facilitate refactoring.

Tool	Topology enhancement	Role of cues	Reference
Crocodile	Creates list of links to code that may need refactoring	Identifies code based on metrics	[Simon et al. 2001]
Soul	Creates list of links to code that may need refactoring	Identifies code based on specific structures	[Mens et al. 2003]
jCOSMO	Displays graph with links to code that may need refactoring	Identifies code based on specific structures	[van Emden and Moonen 2002]
Duploc	Displays navigable matrix (class × class) highlighting clone relationships	Identifies clones based on duplicate lines	[Ducasse et al. 1999]
Eclipse	None; attempts to automate refactoring	Identifies code to transform based on references in abstract syntax tree	http://www.eclipse.org/

automatically scanning for smells and generating a patch of results increases the benefit-to-cost ratio in Equation 1 by reducing C, and as IFT predicts, enables developers to find valuable prey at lower cost. Note, however, that this benefit is

predicated on the tool's ability to approximate (or otherwise obtain) goal *G* accurately; if this is not the case, then the tool's cost reductions will only apply to prey that is not particularly valuable.

3.2.2 Information foraging when performing refactorings

After a developer identifies a refactoring opportunity and selects a particular refactoring, another information hunt begins to answer the question, "What must I do to perform this refactoring?" For example, suppose the developer finds duplicate code and decides to merge two methods, eliminating one. This requires finding and fixing all references to the obsolete method.

In IFT terms, the predator is a developer seeking prey that provide information about what code to modify during refactoring. The topology can again be operationalized as a set of artifacts (e.g., application code, test code, documentation, related emails) linked through navigation mechanisms (e.g., clickable hyperlinks, scrollbars, or menus); the most important prey in this situation are typically application and test code. The cues can be operationalized as data and control references in the code's abstract syntax graph. Such references can be followed to locate the code involved in refactorings [Dudziak 2002].

Existing tools for performing refactorings predominantly aim to fully automate the task. For instance, at present Eclipse can fully automate approximately 60% of refactoring episodes encountered in practice [Xing and Stroulia 2006], substantially reducing the cost *C* of foraging involved in these refactorings. Moreover, new prototype features aim to increase the list of automatable refactorings. In case studies, these prototypes reduced the human effort for refactoring by over 90% [Tokuda and Batory 2001].

These tools rely on the same cues (i.e., data and control references) that developers use to find code to modify when performing refactorings. However, developers rely on human judgment (scent) to discern whether particular references point to code that truly needs modifying. Such judgment is especially important for performing complex refactorings, particularly those that modify the class hierarchy [Tokuda and Batory 2001; Xing and Stroulia 2006]. In these cases, tools may have difficulty accurately approximating the scent that developers perceive; thus, developers might need to review and possibly fix the automated results manually, which adds to their cost *C* of foraging.

Research suggests that automated-refactoring features are more frequently used if they reliably perform *small transformations* and *eliminate the need for foraging*, rather than performing big transformations that the developer must review. For example, one study of 41 Eclipse users found that over half of them used automated refactoring features only for automatically renaming a method/class/variable, moving a method/class (to another class/package), and extracting a new method from duplicate code [Murphy-Hill and Black 2008]. These small refactorings require very little judgment (e.g., when a method is renamed, there is no choice but to modify all referring code to reference the new method's name); thus, accurate goal *G* approximation is relatively simple. On the other hand, only one or two developers used the refactoring features for introducing a factory pattern, pushing down a method from a superclass into a subclass, or introducing a supertype. These refactorings make relatively big transformations to an entire class hierarchy and require judgment, making goal *G* approximation difficult. Thus, developers must often manually confirm that the refactoring occurred properly [Tokuda and Batory 2001; Xing and Stroulia 2006]. (This interpretation of the study described above is also consistent with similar studies using additional data sets [Murphy-Hill et al. 2009].)

As with debugging, these results align well with IFT constructs: A refactoring feature's net worth increases strongly with the amount of foraging cost *C* eliminated by the feature. In particular, the success of these refactoring tools, which aim for full automation, rely heavily on the their ability to approximate goal *G* accurately.

3.3 Code reuse

Whereas refactoring concerns finding existing bad code to improve, reuse emphasizes finding existing good code to use in building new software. Just as finding bad code requires foraging (Section 3.2.1), finding good code also requires foraging (Section 3.3.1), and just as fixing bad code requires foraging (Section 3.2.2), integrating good code into a new context requires foraging (Section 3.3.2). Numerous tools help with these two foraging-centric phases of software reuse (Table 4).

3.3.1 Information foraging when finding reusable code

When faced with a task that might be solved by reusing code, a developer is a predator looking for prey that provide information about what code can solve the problem. The most obvious prey for providing this information is the code itself, although documentation sometimes is useful for identifying reusable code [Ye et al. 2000]. As in debugging and refactoring, prey are embedded in a topology with navigable links provided by the development environment.

Several criteria drive judgments (scent) about whether code is reusable for a particular purpose. The code's functionality must meet the developer's needs [Henninger 1994], and it must have particular quality attributes adequate for the purpose at hand [Caldiera and Basili 1991; Dunn and Knight 1993]. Cues guide these judgments. For instance, identifier names suggest functionality [Henninger 1994], and code structures and code history suggest hints about code quality [Caldiera and Basili 1991].

Without environment support, foraging for code to reuse tends to be a costly process. For example, looking for useful code (prey) in a set of source files, the developer may manually scroll through each file that has a promising name (a cue). If the file mentions the names of other potentially useful classes, the developer may want to inspect those files as well, which may in turn reveal more files to inspect, and so on.

Many search-based tools, such as grep and CodeFinder [Henninger 1994], can reduce the cost *C* of foraging by letting the developer enter a textual query to generate a patch of results [Henninger 1994]. Here, such tools approximate the goal *G* as pieces of directly reusable components that are lexically similar to the query. Accuracy of goal approximation suffers when developers have difficulty coming up with effective query terms—a common problem. To address this difficulty, CodeFinder can suggest alternate query terms [Henninger 1994]. Similarly, a developer can use the Contextual Search tool [Hill et al. 2009] to identify reusable code for extending an existing application. The tool automatically generates queries based on the code that already exists in the application. (This tool actually searches based on automatically generated natural language statements, rather than just keywords.)

Goal approximation need not be based exclusively on lexical similarity. For example, metrics can often reveal whether the structure of code suggests reusability [Caldiera and Basili 1991; Dunn and Knight 1993]. Searching based on code semantics has also been shown to be an effective way to approximate the predator's goal. For instance, the S[6] search tool enables developers to write test cases and contracts (e.g., preconditions and postconditions) that describe the semantics of code that would be suitable for reuse [Reiss 2009]. The tool then searches for code that matches these semantics. Yet another tool, Evolizer [Wursch et al. 2010], enables users to enter natural language questions (with limits on what can be asked) regarding facts about source code. The tool does more than lexical query matching; it infers semantics of the question (e.g., if the question asks about a code structure relationship) to provide an appropriate response.

In addition, the prey need not be directly reusable pieces of code. Sometimes, it is useful just to find code that, although not directly reusable, is "conceptually" reusable in the sense that it can show a developer how to write similar code. For example, Strathcona searches a repository of example code (e.g., examples of how to call and combine API functions) to find code snippets previously used in applications that were structurally similar to the application where the developer is currently working (Figure 5a) [Holmes et al. 2006]. Strathcona uses predominantly type-related similarities to guide this search (for example, analyzing inheritance relationships to assess similarity). The search results, in turn, are annotated with cues explaining the rationale for why each link appears in the results (Figure 5b). These cues aim to engender accurate scent S_{ji} in the mind of the developer.

Table 4. Examples of tools that facilitate code reuse

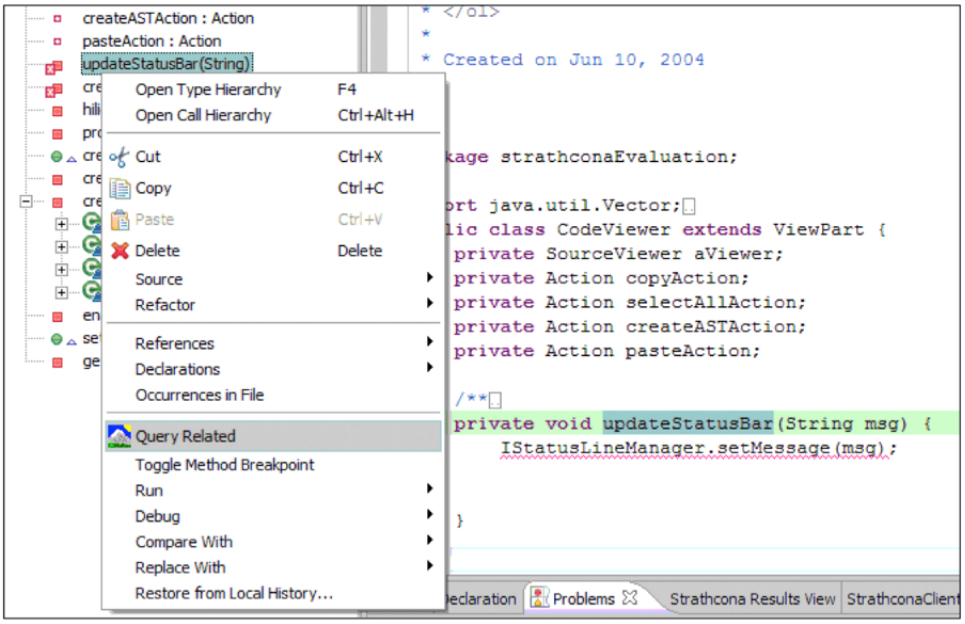
Tool	Topology enhancement	Role of cues	Reference
CodeFinder	Creates links to code that might be reusable for a particular problem	Uses lexical cues to identify code	[Henninger 1994]
Code Miner	Creates links to code that might be reusable for a particular problem	Uses structural cues calculated via metrics to identify code	[Dunn and Knight 1993]
Strathcona	Creates links to code that might be reusable for a particular problem	Uses structural constraints to identify code; annotates each link with the rationale for its inclusion in the search results	[Holmes et al. 2006]
Gilligan	Creates links to dependencies that might need to be extracted during integration	Identifies dependencies based on references in the abstract syntax tree; allows developers to add new cues in the form of annotations	[Holmes and Walker 2007]

As we have seen with both debugging and refactoring tools, the success of these search-based tools hinges on their ability to accurately approximate the predator's goal *G*. Studies have demonstrated that these successful tools possess this ability. For example, one study showed that Strathcona outperformed grep (and Eclipse's find utility) at delivering relevant and useful results to developers, thus helping the developers find reusable code more efficiently [Holmes et al. 2006]. Another experiment used Code Miner to find reusable code in five applications; when human experts rated the results, more than half of the recommendations scored highly [Dunn and Knight 1993].

3.3.2 Information foraging when integrating reusable code

Having located relevant code for reuse, the developer must adapt it for integration into another context, initiating another round of foraging to answer the question, "What must be done to integrate this code?" Often, the answer is that the reusable code must be modified or additional code must also be extracted. For example, other code might be needed to pass data to the desired code, or might be needed because it

(a) The developer selects the incomplete method, `updateStatusBar`, and initiates a query to Strathcona.



(b) Strathcona returns a patch of three relevant examples. Each example is labeled with a textual cue indicating Strathcona's rationale for choosing it.

Rationale	Artifact
Class has parent of type	org.eclipse.ui.part.ViewPart
Method Calls Target Method	org.eclipse.jface.action.IStatusLineManager.setMessage(java.lang.St
Class uses Class	org.eclipse.jface.action.IStatusLineManager

Fig. 5. The Strathcona Eclipse plug-in. (Images courtesy of Reid Holmes)

is called by the component of interest. However, judgment is needed, because it is not always necessary to extract all these peripheral components. Making this judgment requires finding information about the reusable code's dependencies and considering what code edits could eliminate dependencies.

In IFT terms, the primary prey during integration is the code on which the previously identified reusable code depends. The cues for finding dependencies can be operationalized as references in the code's abstract syntax graph representation. For example, when reusing a class, an inheritance relation signals that the reusable class depends on a parent class; thus, information about the parent may be needed to decide whether to extract or replace the parent. Furthermore, the parent class may contain cues that indicate additional dependencies (prey) for the developer to consider.

One class of tools seeks, in theory, to obviate foraging, thus reducing cost C to 0. These tools succeed or fail by the accuracy of their goal G approximations. Given a component to be reused, they automatically find all dependencies by following references (e.g., extracting code with a code-slicing algorithm [Lanubile and Visaggio 1993]). Tools, such as Jigsaw [Cottrell et al. 2008], automate the modification of extracted code to fit a new context. Although full automation might be desirable, it does not seem feasible because it rests on totally accurate goal approximation. Thus, many tools include user interfaces to obtain the significant amount of developer judgment needed about how to handle dependencies and how to modify extracted code. Even with these limitations, studies show that Jigsaw successfully eliminates the most time-consuming aspects of reuse (i.e., reducing C) and is effective at highlighting code otherwise overlooked by developers [Cottrell et al. 2008].

Gilligan provides additional support for foraging during integration of reused code [Holmes and Walker 2007]. First, Gilligan enhances the topology by adding links between dependencies, creating a patch of components that can be quickly navigated. Then, Gilligan supports enrichment by enabling developers to manually mark dependencies with annotations that record whether or not dependencies should be extracted. These annotations can later serve as cues if a developer needs to revisit the integration process. These custom cues aim to suggest highly accurate scent S_{ji} to developers looking for code related to reuse decisions. Moreover, Gilligan color-codes the cues, reducing the time it takes for developers to focus their attention W_j on the relevant cues.

IFT predicts that if developers' expectations in Equation 2 are closely aligned with the actual benefits and costs in Equation 1, then they will make navigation choices that lead to more prey at less cost. Studies of Gilligan are consistent with this prediction, showing that helping developers to enrich the environment with new cues can provide them with the means to more effectively forage for information and to improve their productivity overall. In particular, four case studies showed that Gilligan reduced the cost of reuse tasks, and thus, helped developers perform larger reuse tasks than they would normally attempt [Holmes and Walker 2007].

4. IFT-BASED PATTERNS FOR SOFTWARE ENGINEERING

The results of our assessment of software engineering tools suggest that we can abstract IFT-based design principles from specific tool designs to patterns. This allows us to view the commonalities among tools in a way that is not only tool independent, but also independent of any particular software engineering area.

A design pattern is a general, reusable solution to a common design problem [Borchers 2000; Gamma et al. 1994]. By succinctly capturing design guidance in an abstract—yet practical— form, patterns have proven useful in many software engineering endeavors, including research aimed at designing environments for programming [Pane 2002] and visualization [Hundhausen 2005]. In industry,

developers rely on patterns when designing APIs [Bloch 2001], service-oriented architectures [Erl 2007], and object-oriented systems [Gamma et al. 1994]. Benefits of design patterns include enabling participatory design in which users actively engage in design, supporting design debates by providing a technical lexicon, serving as organizational memory that persists beyond individual memory, supporting communication between designers, and providing rationale for design decisions [Dearden and Finlay 2006].

Following in the tradition of the "gang of four" [Gamma et al. 1994], we worked bottom up to identify twelve *information foraging design patterns*, pulling directly from design solutions. Viewing SE tools through an IFT lens, we noted similar solutions to recurring information foraging problems. We then identified these solutions' distinguishing characteristics relating to IFT's constructs (see Table 1). This process for identifying patterns is common among pattern researchers (e.g., [Alexander 1979; Gamma et al. 1994; Tidwell 2006]); the key distinction of our process was the IFT perspective.

The remainder of this section describes the information foraging design patterns that we abstracted from the tools in Section 3. This set of patterns is not intended to be exhaustive, but rather to serve as a starting point for future research aimed at applying, extending, and augmenting the set. Similar to other work documenting design patterns (e.g., [Gamma et al. 1994; Tidwell 2006]), we describe each pattern using a template that comprises the pattern's name, a terse description of the pattern's purpose (*Intent*), a motivating example of the pattern in use (*Motivating Example*), an abstract description of how the pattern plays out (*Description*), a characterization of situations where the pattern is appropriate (*Applicability*), the pattern's benefits and liabilities (*Consequences*), cases where the pattern has appeared (*Known Uses*), and relationships to other patterns (*Related Patterns*).

4.1 Patterns for Identifying Valuable Prey and Patches

What information in the environment constitutes valuable prey to the current developer, and what patch is it in? In this section we present six patterns, derived from multiple existing debugging, refactoring, and reuse tools, that identify prey and its patches.

Since patterns for identifying prey have essentially the same benefits, liabilities, and relationships to other patterns, we present Consequences and Related Patterns sections for all the prey-identifying patterns here:

Consequences

Benefits: Prey-identifying patterns can reduce the cost of foraging (or obviate the need for it altogether) by making valuable patches available to the developer.

Liabilities: If the identified patches are not actually valuable to the developer, then the developer may waste time investigating the patches and get little benefit.

Related Patterns

Prey-identifying patterns can be used in combination with patterns concerned with presenting prey to the developer (i.e., those in Section 4.2). For instance, the Structural Relatedness pattern, which identifies potentially valuable patches that have structural connections to the current patch, can be used in combination with the Gather Together pattern to collect the identified patches into a summary patch that enables the developer to inspect those patches with little effort.

4.1.1 Expertise Recommender

Intent

To identify people who can help the developer with a particular information goal.

Motivating Example

A developer wants to modify code within a component. Before making the modifications, he may want to contact the owner of that code to coordinate changes or to understand the program. The code owner is considered an expert in this particular part of the system and can therefore help a developer fulfill his information goal set. A tool could track which developers are experts on which code, and tell the developer who the relevant experts are.

Description

The Expertise Recommender pattern identifies if the predator may want to communicate with someone that can help the predator locate information. An expertise recommender tool provides cues that reduce the cost of contacting the person who fulfills a developer's information goal set by (a) providing knowledge that the person in fact may be able to help and (b) by providing contact information.

Applicability

Because the Expertise Recommender pattern is intended to help a developer find a person, it can be used when all the following conditions are true:

- there are patches that contain the contact details (e.g., email address or phone number) for people who have information that a developer needs
- it is possible to index these people according to the knowledge or other expertise that they have
- it is possible for the developer to indicate the needed information, either explicitly or implicitly
- when the developer indicates needed information, a set of links can be displayed to reach the patches containing contact details for the people who have that information

Known Uses

Expertise Browser by Mockus [Mockus and Herbsleb 2002] is an expertise recommender that identifies expert developers based on the changes they made to code or documentation. Ye et al. [Ye et al. 2007] has also developed a recommender that identifies a person's expertise based on recently modified code, and recommends that person as an expert to others.

Emergent Expertise Locator [Minto and Murphy 2007] uses call dependencies as well as frequency of code modification to identify people who develop code that may be related to the programmer's code.

4.1.2 Lexical Similarity

Intent

Identify patches that contain valuable prey based on their lexical similarity to words that emit strong scent to the developer.

Motivating Example

Consider a new developer on a large, well-established web server project who is trying to understand the cause of a subtle bug in one of the classes. The developer faces a daunting (and time consuming) task in extracting the relevant information from the project's massive collection of artifacts, which includes a code base of over a million lines of code and an archive of over fifty thousand e-mail messages.

The Lexical Similarity pattern can help in this situation by automatically searching for other artifacts related to the class that might shed light on why the class is defective. In particular, the pattern identifies artifacts that contain the same words as the class' definition and therefore are likely related to the class. For instance, the pattern can search the project's e-mail archive for messages with similar words to the class, and therefore, likely discuss the class. If the bug is recurrent (i.e., previously fixed and then accidentally reintroduced), such messages might contain valuable discussion of how the bug was originally fixed.

Description

The Lexical Similarity pattern identifies the lexical properties of each patch in the environment. For instance, this may involve knowing the words contained in the text of each patch and their frequencies. It is common to ignore stop words, short frequently occurring words, such as *the* and *or*. When dealing with code text, it is also common to treat programming reserved words as stop words and to break "camel-case" words into their constituent parts (e.g., *helloWorld* can be broken into *hello* and *world*).

The pattern defines a function that maps from a set of words that describe desired prey to an ordered set of other patches such that the patches are ordered from most related to least related based on the patches lexical properties. The words describing desired prey may be explicitly elicited from the developer (e.g., via a textual query field) or implicitly inferred from the developer's behavior (e.g., by looking at the words in the methods that the developer chooses to visit). It has been common among tools that use this pattern to use techniques from information retrieval [Baeza-Yates and Ribeiro-Neto 1999] to perform the mapping (e.g., by combining tf-idf and cosine similarity).

Applicability

The Lexical Similarity pattern is based on the assumption that if a developer finds a patch with certain textual features valuable, then other patches with similar textual features will likewise be valuable. Therefore, this pattern can be used when all the following conditions are true:

- valuable patches contain textual features
- the value of a patch is highly related to which textual features it contains
- the text in patches can be indexed
- when a developer visits a particular patch, it is possible to display links to a set of patches with textually-similar features

Known Uses

Traditional code search tools that allow developers to enter natural language queries (e.g., Eclipse's search utilities, the CodeFinder tool [Henninger 1994], and the Hill et al.'s contextual search tool [Hill et al. 2009]) use the Lexical Similarity pattern, often augmented with additional features, for example, for eliciting high-scent words from developers that coincide with high-value patches. Evolizer [Würsch et al. 2010] also supports developers with natural language queries, but uses an ontology language for

its backend. The Hipikat tool [Cubranic et al. 2005] uses lexical similarity to a selected artifact (or portion thereof) to identify related artifacts, possibly of different types (e.g., code, bug reports and version control log entries). Exemplar [McMillan et al. to appear] uses API documentation to search entire projects on SourceForge for potential code examples.

Debug Advisor [Ashok et al. 2009] is another "lexical similarity" pattern tool. It uses tf-idf to search debug reports (with stack traces) to identify similar and related debugging traces.

4.1.3 Past Aggregate Behavior

Intent

Identify patches that contain valuable prey based on the past behavior of developers working on the project.

Motivating Example

Consider a developer who has recently joined a software project and is tasked with fixing a defect in one of the software's features. The developer has managed to find one method related to the feature, but is still faced with the daunting task of finding and comprehending all the other methods relevant to the feature.

The Past Aggregate Behavior pattern offers a way to assist this developer. A tool based on the pattern could, for example, take advantage of the project's version control log. Developers who work on a feature will tend to make commits that touch the code related to the feature [Zimmerman et al. 2004]. Thus, the tool could analyze the commit log, identifying methods that were changed during the same commits as the developer's current method and are thus likely to be relevant to the feature.

Description

The Past Aggregate Behavior pattern presumes that valuable prey is prey that other predators have recently hunted for. The pattern identifies valuable prey by employing a *logger* to maintain a history of project developer accesses or modifications to patches in the environment. The logger aims to record history over the entire history of the project and over all developers working on the project. The pattern also defines a *coupling function* that maps from a current patch in the environment to closely related patches based on the history. The coupling function commonly uses temporal proximity of accesses to patches as an indicator of patch relatedness.

Applicability

The Past Aggregate Behavior pattern depends on the assumption that the current developer's prey is probably in a place that other developers have frequently accessed. Therefore, this pattern can be used when all the following conditions are true:

- it is possible to identify a set of developers who have similar needs for prey
- most developers generally find the prey that they need
- logs of those developers' navigations can be collected and aggregated
- when a particular developer needs to find prey, the aggregate navigation of other developers with similar needs can be displayed, with links to access patches frequently accessed by other developers

Known Uses

For a given project, the Team Tracks tool [DeLine et al. 2005] logs the methods that developers visit (based on text cursor position) and assumes that methods visited in close succession are related. Given a selected method definition, Team Tracks will identify the most related methods.

The ROSE tool [Zimmermann et al. 2004] treats the project's version control logs as a record of past behavior and analyzes the logs to extract *evolutionary coupling* relationships (i.e., relationships in how code entities were modified). Given a particular change made by the current developer, ROSE identifies additional entities that the developer may need to change.

4.1.4 Personal Working Set

Intent

Identify patches that the developer will likely revisit based on how recently the developer accessed or modified patches.

Motivating Example

Developers revisit methods frequently while working on object-oriented code [Parnin and Gorg 2006, Piorkowski et al. 2011]. In information foraging terms, they may revisit a method (i.e., patch) to find prey that they missed on a previous visit or to link to other methods. Because developers revisit methods so frequently, reducing the cost of revisit navigations can lead to a considerable reduction in the developer's overall cost.

The personal working set pattern can help reduce the cost of revisits by keeping track of the most recently visited locations in the code. For instance, a tool might log accesses and edits to methods and use the log to identify methods that developer seems most likely to revisit based on her past behavior.

Description

The Personal Working Set pattern prioritizes prey that is frequently visited while the developer works on a task. The pattern uses a *logger* to record a personal history for a particular developer. The history includes developer actions, such as clicks within patches and/or modifications to patches. The pattern defines a *relevance function* that maps from the history to an ordering of the patches in the history from most likely to be revisited to least likely. The relevance function often relies on a combination of how recently the developer accessed a patch and the level of interest that the developer exhibited while accessing the patch. The logger may capture the developer's entire history with the project or may partition the history by subtask. In the latter case, the pattern implicitly uses the task of interest to focus in on patches that contain valuable prey.

Applicability

This pattern reduces the cost for a developer to revisit patches, so it can be used when all the following conditions are true:

- each developer frequently needs to return to a patch previously visited
- the cost of returning to a patch is much higher than the cost to reach it initially
- the set of patches visited by a developer can be logged

Known Uses

The Mylyn tool [Kersten and Murphy 2006] logs code entities that a developer selects or edits to produce a degree of interest model. The more an entity is selected or edited, the more interest is inferred. Mylyn uses this model to identify the entities that the developer is most interested in and therefore most likely to revisit.

Parnin and Gorg [Parnin and Gorg 2006] developed a recommendation system based on programmer navigation. The recommendation system included methods the developer had already visited. To determine which methods were most relevant, Parnin and Gorg tried a variety of algorithms to determine the best way to operationalize working set.

4.1.5 Structural Relatedness

Intent

Given a current patch in which the developer has indicated interest (e.g., by navigating to the patch and/or modifying the patch), this pattern seeks to identify other patches that contain valuable prey by looking at other patches that are structurally related to the current patch.

Motivating Example

Consider a developer who sees that a cell in his graphical spreadsheet application is displaying an incorrect value. He would like to find the code that set the value being displayed; however, the code base is extensive, and there many places in the code that manipulate the variable that holds the value to be displayed.

A tool could assist this developer by identifying the lines of code that caused a particular observable output. Providing the developer with the lines would save him considerable effort in locating them by, for instance, manually tracing dependences in the source code.

Description

The Structural Relatedness pattern maintains a model of the structural relationships between patches for a particular type of relationship(s). For instance, the model might be represented as a graph with the vertices as patches and the edges as the structural relationships between the patches. Using this representation, the pattern can map from a particular patch to the other patches that are structurally related by traversing edges.

Applicability

This pattern depends on the assumption that if a developer finds a patch to be useful, then patches that are structurally related (e.g., through method calls) will likewise be valuable. Consequently, this pattern can be used when all the following conditions are true:

- valuable patches are connected to one another via structural relationships
- it is possible to parse or otherwise analyze the patches in order to identify structurally-related patches
- when a developer visits a particular patch, it is possible to display links to a set of structurally-related patches

Known Uses

Eclipse uses the Structural Relatedness pattern for its Open Call Declaration and Call Hierarchy features. For the former, it uses declaration dependencies to identify valuable patches, and for the latter, it uses call dependencies.

The Gilligan tool [Holmes and Walker 2007] uses a variety of code-structural relationships, including inherited-from, calls, references, declared-by, and contained-by, to identify valuable patches for developers.

The Whyline tool [Ko 2008] uses the Structural Relatedness pattern to, given an observable program output, identify the code that directly caused that output. Moreover, Whyline uses data and control dependences to identify the code that caused a line of code to execute or exhibit a particular behavior.

Information Fragments [Fritz and Murphy 2010] enable developers to ask about the relationship between two "fragments" or entities in a software repository. Information Fragments explore relationships among code entities that are not necessarily code-specific but are still related through structure. Some examples include “Dev 2 is assigned to work item A” and “Test case 1 tests Code X”.

4.1.6 Task Heuristic

Intent

Identify patches that contain valuable prey by leveraging knowledge about the specific task that the developer is performing, and in particular, about what information developers typically seek during the task.

Motivating Example

Consider a developer tasked with looking for code to improve by refactoring. One way that developers commonly find code in need of refactoring is through the identification of bad smells. However, finding bad smells may involve systematically inspecting a code base—an overwhelming task for large systems.

The Task Heuristic pattern can be applied to automatically identify code with bad smells, thus saving the developer considerable effort. For instance, a tool based on the pattern could look for unusually large classes to identify the “large class” smell [Fowler and Beck 1999].

Description

The Task Heuristic pattern uses information from the environment or elicits information from the developer in order to help that developer accomplish a task, such as refactoring code or removing clones. The pattern defines a *matching heuristic* that takes as input features in the environment, such as patches, and return whether the features contain prey relevant to the task type. The pattern can apply the matching function over the entire environment to, for instance, identify all patches that contain valuable task-specific prey.

Applicability

This pattern is based on the assumption that automatically computable metrics can identify patches relevant to a particular task. Different tasks might have different metrics, and developing such metrics can require substantial research. Consequently, the Task Heuristic pattern can be used when all the following conditions are true:

- a particular task requires performing costly navigation to patches
- automatically computable metrics can accurately identify patches relevant to the task

- this task is done by enough developers, and/or done frequently enough by developers, that it is desirable to discover and refine metrics for identifying task-relevant patches
- when the developer needs to perform the task, a set of links to relevant patches can be displayed

Known Uses

Refactoring tools apply this pattern in a variety of ways to identify code in need of refactoring (i.e., prey to a developer looking for refactoring opportunities). Crocodile [Simon et al. 2001] uses complexity metrics to identify overly complex code. Soul [Mens et al. 2003] and jCOSMO [van Emden and Moonen 2002] identify code with structures indicative of bad smells. Duploc [Ducasse et al. 1999] computes similarity between classes to identify code clones.

Similarly, reuse tools apply the pattern to identify reusable code. For instance, CARE [Caldiera and Basili 1991] and Code Miner [Dunn and Knight 1993] use code metrics to identify reusable code. As another example, the S^6 tool [Reiss 2009] uses test cases to identify code to reuse that has the desired semantics.

4.2 Patterns for Presenting Prey

Once the prey is identified, many tools help developers locate and keep track of this prey, with an emphasis on reducing the cost *C* of doing so. In this section we present six patterns derived from multiple tools that reduce cost *C* through manipulation of the *environment* in which a software developer works.

4.2.1 Community Portal

Intent

To provide an information patch where multiple contributors can enrich the patch with information features, especially cues.

Motivating Example

A new developer wants to implement a feature in an open-source software project, and is trying to learn the system's architecture.

Rather than searching for documentation, the new developer instead discovers the original developers' Wiki. In this Wiki, other developers have provided examples and tutorials to help a newcomer understand the system.

Description

The purpose of the Community Portal pattern is to provide an information patch that are primarily populated with information features that other foragers have found useful. The Community Portal pattern requires that a forager be able to modify the information patch to add information features and links.

The other foragers that are enriching this space should have a similar profile as the forager who is searching the space. That is, if the predator is a developer, for the Community Portal to be useful, the other foragers should also be developers and should be posting information that is relevant to a developer.

Applicability

This pattern is based on having developers provide useful cues to one another through a central information patch. Therefore, it can be used when all the following conditions are true:

- a developer needs information that other developers know how to find
- the other developers are willing to help the developer who is need
- the other developers can proactively anticipate what information will be useful, or else it is possible for a developer to express a need and then wait for other developers to respond
- a central patch can be created, where developers can post useful information and/or requests for help

Consequences

Benefits: This pattern reduces the developer's cost of navigating to potentially useful prey. The rationale is that prey that is useful to others could also be useful to the developer. In addition, a team of developers is more able to discover potentially valuable information patches, and by keeping the links in one information patch, the cost of navigating to those patches is lowered. Also, any links created are not specific to one domain, allowing developers to potentially create links to multiple sources of information in one patch.

Liabilities: The pattern heavily relies on other foragers. If only a few foragers enrich the information patch, then the developers who use the portal may not see the information in it as valuable. In addition, it requires that those who enrich the portal have a similar view of what is "valuable" information. The data must also be kept fresh in order to be relevant. If the content in the Community Portal does not accurately reflect the software system, then it will not be valuable to the predator.

Known Uses

Rational Team Concert contains a community portal [Treude and Storey 2011] where developers can access documentation. Also, development-centered Wikis [Dagenais and Robillard 2010] also serve the purpose as a community portal. IBM Lotus Notes and Sharepoint are other systems that can be used as a Community Portal [Lanubile et al. 2010].

StackOverflow [Stack Exchange Inc. 2011] is a questions-and-answers web site where other users answer questions. A rating system causes valuable answers to become accessible. On average, 42% of the questions posted on StackOverflow receive an acceptable answer [Treude et al. 2011]. Community experts are providing an accessible archive that others can use as a starting point for information.

Related Patterns

This pattern is similar to the Signpost pattern because both patterns enable predators to enrich the environment. However, signposts are merely markers, whereas a Community Portal's information features are links to other information patches.

4.2.2 Cue Decoration

Intent

To automatically change the appearance of a cue to attract more of the developer's attention W_j or augment a cue with additional information to improve the accuracy of scent S_{ji} discerned by the developer.

Motivating Example

Consider a developer performing a maintenance task on a large code base. The developer encounters a call to an unfamiliar method, and must decide whether or not to expend the effort to inspect the method. Unfortunately, the only information

available about the method is its name, which happens to be poorly chosen and uninformative, hampering the developer's ability to discern accurate scent.

To address this problem, many modern development environments, such as Eclipse, decorate method names in the code editor with tooltips. These tooltips contain snippets of documentation that provide additional information about methods, aiding the developer in accurately assessing scent S_{ji} and bringing the developer's expectations of the value to be gained by following a link E(V) more in line with the actual value V.

Description

The Cue Decoration pattern changes the appearance of a cue to influence the terms in Equation 3. For example, color, size, animation, iconography, and font boldness can be changed to affect how a predator establishes scent S_{ji} or to draw the predator's attention W_j. Cues can also be augmented with tooltips or other new text. For example, if a tool creates a new navigable link in an environment, it can provide a textual annotation explaining why that link was created.

Applicability

This pattern enhances cues so that they are more prominent or informative, so it can be used when all the following conditions are true:

- the information environment contains many cues that threaten to distract the developer away from valuable patches or cues that do not convey the relevance of their associated links
- it is possible to automatically generate enhanced cues that are capable of informing developers about whether a distal patch is valuable
- the enhanced cues can be displayed when a developer takes an interest in a patch or link to that patch (e.g., by showing auto-generated tooltips on mouse-over)

Consequences

Benefits: Cue decoration can increase developers' ability to estimate the value V to be obtained from a particular distal patch, thereby bringing Equation 2 closer into alignment with Equation 1 and enabling developers to make decisions that lead to a higher ratio of value to cost. Cue decoration can also attract the developer's attention to particularly relevant cues by increasing W_j.

Liabilities: Too much decoration can overwhelm developers, making it difficult for them to know where to direct their attention. Misleading decorations can cause developers to discern inaccurate scent, causing the value of Equation 2 to be ill aligned with that of Equation 1.

Known Uses

Eclipse decorates class and method identifiers with tooltips displaying documentation. Jigsaw decorates code with color highlighting to indicate how integration was performed after automatically integrating reused code into a new application, [Cottrell et al. 2008]. Duploc uses color to highlight similarities and differences between two purported clones, making it clearer whether both code segments need attention during a particular task [Ducasse et al. 1999]. Hipikat and Team Tracks decorate links' labels with text and icons, respectively, to indicate why each new link was created [Cubranic et al. 2005; DeLine et al. 2005].

Related Patterns

The Cue Decoration pattern can decorate cues created using any one of the prey identification patterns. The Cue Decoration pattern may be used to enhance cues in the Dashboard, Community Portal, or Gather Together patterns.

4.2.3 Dashboard

Intent

To generate an information patch in which a developer can become aware of links that lead to continually changing information patches relevant to his or her work.

Motivating Example

A developer who is managing many different tasks simultaneously must be able to keep track of the tasks, their dependencies, and incoming events. Rather than having the developer periodically check on each task by navigating to and inspecting the various task-relevant artifacts, the system could instead provide a single interface that provides up-to-date task information.

Description

The Dashboard is an information patch that contains up-to-date information relevant to the developer. The Dashboard is automatically updated to reflect the latest events occurring within a development project. Information features shown in the Dashboard may include incoming bug reports, approvals for checking code modifications into a source repository, and incoming comments on issues.

Applicability

The Dashboard pattern creates a patch that summarizes the current status of other patches. Consequently, it can be used when all the following conditions are true:

- on an ongoing basis, a developer needs to be aware of the status of multiple patches
- the status of these patches changes somewhat unpredictably, making it necessary for the developer to repeatedly check the status of those patches
- it is costly to navigate to each of these patches
- the developer can identify these other patches ahead of time
- it is possible to succinctly summarize the status of each of these patches
- a navigable display can be shown, which presents these summaries along with navigable links to each distal patch

Consequences

Benefits: The developer does not have to specify what information that the Dashboard gathers whenever he or she wants to use it. Instead, when the developer accesses the Dashboard, it updates itself to reflect the latest events. A Dashboard can serve as a "launching point" from where the developer plans the day's tasks or determines the day's information goals.

Liabilities: A Dashboard must gather information specific to the developer to be useful. If it presents information that is not relevant to the developer, the developer may waste time processing the information with little or no benefit. Similarly, if a Dashboard presents out-of-date information (e.g., because it does not update frequently enough), then the developer may waste time processing information that is invalid.

Known Uses

IBM Rational Team Concert provides a personalized location in which developers can access their personal RSS feeds [Treude and Storey 2010]. This feed shows a developer currently assigned tasks and notifications—such a developer dashboard provides a central location that is automatically kept up-to-date. Many issue-tracking tools, such as Bugzilla and JIRA, contain similar views that enable a developer to monitor incoming assignments or outstanding issues.

Related Patterns

The Dashboard is highly related to the Gather Together pattern because both patterns involve the generation of an information patch. The primary difference with the Dashboard is that the information in the Dashboard is usually automatically updated, and is focused usually on a single global task, whereas the Gather Together pattern is used for a variety of tasks.

4.2.4 Filtering

Intent

To enable the developer to filter out irrelevant information features from a patch, reducing the cost of processing the patch.

Motivating Example

Because software systems are complicated, many aspects of the source code are not relevant to a developer at a given point in time. A tool can help the developer by hiding irrelevant information, enabling the developer to focus his time and attention on the relevant information.

Description

The Filtering pattern enables a developer to remove features from an information patch. The filtering may occur based on explicit input from the developer, or may be inferred by the tool based on the developer's interaction with the environment.

Applicability

The Filtering pattern enables developers to remove features from patches, so it can be used when all the following conditions are true:

- patches contain a large number of features that are not valuable to a developer
- the developer can express rules that distinguish valuable features from less valuable features
- the features that do not meet these rules can be removed

Consequences

Benefits: The pattern reduces the cost of processing an information patch by eliding information feature of little or no value.

Liabilities: The pattern may inadvertently remove valuable prey, thus depriving the developer of that prey.

Known Uses

Eclipse enables filtering on source code in the form of folding, which allows a developer to hide code such as methods, comments, classes, and package lists. Mylyn filters the Eclipse package explorer view based on a developer's working set. An

issue-tracking system, such as Bugzilla, enables the developer to use advanced search features that filter the search results by the developer's criteria of interest.

Related Patterns

Prey identification patterns can be used as criteria for filtering, and may possibly automate the filtering for the predator. For example, following a developer's working set would be one way to show only relevant items to a developer.

4.2.5 Gather Together

Intent

To enable a developer to assemble information features from disparate patches into a single patch, thus reducing the cost of navigating between those features.

Motivating Example

Consider a developer who is studying code to which he wants to add new features. There are many interacting methods that the developer needs to comprehend, but he must navigate between different files, methods, and views to inspect and understand the relevant code fragments.

A tool could reduce this cost by enabling the developer, as he identifies relevant methods, to gather those methods in a single workspace (i.e., patch). Once the developer has gathered the methods, he can conveniently inspect their interrelationships without the need for costly navigation.

Description

The Gather Together pattern enables the developer to gather information features into a customized information patch. The customized patch acts as a view of the information features of the source patches. That is, gathering does not modify the source patches. Changes to the information features in the source patches are reflected in the customized patch, and vice versa.

Applicability

The Gather Together pattern enables the developer to manually collect information required for a particular task. Thus, this pattern can be used when all the following conditions are true:

- completing a task requires the developer to combine information from multiple patches
- it is not possible simply to remember this information (e.g., because a very large amount of information must be gathered, or finding each piece of information is so slow that the developer might forget the information that was already found)
- the developer can be provided with a way to create a new patch, then to copy information from other patches as they are found

Consequences

Benefits: The pattern enables the developer to assemble information from throughout the environment into a single patch, eliminating the need for him or her to navigate between patches while processing that information.

Liabilities: The pattern depends largely on the ability of the developer to identify information that will be relevant without fully processing it. If the developer cannot

guess what information will contain relevant prey, then he or she may still need to navigate outside the customized patch to track down information.

Known Uses

Jasper [Coblenz et al. 2006] enables a developer to display multiple related methods using a compact view. Code Bubbles [Bragdon et al. 2010] enables a developer to assemble an information patch with code that is in his or her working set. Gaucho [Olivero et al. 2011] enables the developer to arrange succinct visualizations of code fragments within 2-D workspaces.

Related Patterns

Any of the prey identification methods can be used to facilitate gathering. For example, lexical similarity or structural relationships may be used to identify code related to another fragment of code. The Gather Together pattern can also be applied in combination with the Filtering pattern, which can be used to filter the resulting view. The Dashboard pattern is similar to the Gathering Together pattern in that it also involves a customized information patch; however, the Dashboard pattern focuses on helping the developer to maintain awareness, rather than to gather information for a specific task.

4.2.6 Signpost

Intent

To support enrichment by enabling the developer to leave cues in the environment which, when seen later, will attract the developer's attention W_j and/or engender accurate scent S_{ji} in the predator's mind.

Motivating Example

Consider a developer trying to decide whether to reuse a particular method from an existing application. The developer needs to assess whether the cost of extracting the method exceeds the cost of rebuilding it. This assessment involves inspecting the code's dependencies, which requires navigating to dependencies, examining their relationship to the primary code of interest, and recursively exploring more and more dependencies.

SE tools, such as Gilligan, can help by enabling the developer to record decisions along the way [Holmes and Walker 2007]. In particular, developers can annotate dependencies with accept/reject tags. When subsequently performing the extraction, the annotations serve as cues that catch the developers' attention and direct them to the dependencies to be extracted.

Description

The Signpost pattern enables the predator to explicitly add cues to the environment. The predator intends for each cue to engender highly accurate scent when subsequently encountered. These cues may be used by a single predator or shared among a group.

Applicability

This pattern enables the developer to add cues to an information patch, so it can be used when all the following conditions are true:

- an information patch is likely to be visited again (either by the same developer or by others)

- the developer can anticipate what cue might be useful upon this later visit
- the developer is willing to create that cue
- the cue can be collected from the developer and stored in a persistent location, then displayed when the patch is visited later

Consequences

Benefits: Manually encoding information in cues can ease future foraging by reducing the need to relearn that information and, thus, by engendering accurate scent in the mind of the developer. In other words, the signpost cues help align the value of Equation 2 with that of Equation 1. Signpost cues may also attract the predator's attention (i.e., increased W_j) because the predator recognizes the cues, having created them.

Liabilities: If the deposited cues are insufficiently expressive or are ill chosen, predators may register inaccurate scent. Thus, the value of Equation 2 may stray far from that of Equation 1. If predators leave too many signpost cues in the environment, they may have difficulty determining which cues to focus on.

Known Uses

TagSEA enables a developer to write customized text throughout different locations of the code with a searchable annotation [Storey et al. 2007]; Rational Team Concert has a similar capability that developers use within bug-tracking systems as well [Treude and Storey to appear]. Gilligan [Holmes and Walker 2007] uses the Signpost pattern to support developers' foraging for reusable code. Given a candidate reusable method, the tool enables the developer to annotate the method's dependencies with accept/reject tags. Mylyn enables developers to label code with tags that indicate what tasks involve the code [Kersten and Murphy 2006].

Related Patterns

The Signpost pattern may be used in combination with the Gather Together pattern to annotate a generated patch. The Signpost pattern is distinct from the Cue Decoration pattern in that developers manually set signpost cues, whereas cue decoration is automated. Although the Community Portal also involves enrichment of an information patch, that pattern is about leaving information features and links, rather than just cues.

5. FUTURE RESEARCH OPPORTUNITIES

Each term in IFT's core equations can reveal opportunities for SE approaches to help developers during information-intensive software engineering activities. The preceding section's patterns focus on C and E(V), but future research could target the other terms in these equations, namely E(C) and V. In addition, future research could target C and E(V) in novel ways (e.g., through new visualizations or other novel presentations). In this section, we point to opportunities in both of these directions.

5.1 Opportunity: Raising information value V of an existing patch

None of the patterns above takes an *existing* patch and automatically raises its information value V. This would mean automatically augmenting the information content of a patch (e.g., automatically increasing the information content of a method). Perhaps the tools that come closest to automatically raising the information value of a patch are those that allow developers to manually add annotations to code (e.g., Gilligan [Holmes and Walker 2007]), but this process is not automatic. In contrast, the Cue Decoration pattern can be used to automatically add information to pages; however, the pattern is mainly concerned with attracting developers to follow

links to *other* patches that have the greatest ratio for Equation 1. One possibility for increasing an existing patch's value V is automated annotation systems, which proven empirically useful in fields outside of software engineering (e.g., automated image annotation [Jeon et al. 2003] and automated data annotation [Arlotta et al. 2003]). Similar strategies might be useful in software engineering for increasing the value V of patches.

For example, the patch might be a SourceForge project page, which describes a component available for reuse and offers a link to download the source code. The value of this page could be increased by providing information content that developers would benefit from knowing when making a decision whether to reuse the component. For instance, developers might want to know whether the component is interoperable with other components that they have already chosen to use in an application (since components sometimes embody conflicting assumptions about threading and other issues, resulting in architectural mismatches that lead to delays and bugs [Garlan et al. 2002]). In addition, developers might want to know data about specific quality attributes, such as performance and reliability. Such information might be synthesized out of log data, for example by intelligently mining what components fail to operate well together or by logging performance data and reporting it back to the SourceForge server for integration into the project page.

This illustration offers at least one idea for how to enhance the information value of a patch. This tool first would synthesize raw data that is not readily accessible by developers, or perhaps is even not collected at all. The tool would then display the synthesized information in a patch (e.g., as annotations or charts). Future research could identify additional questions that developers ask, extending the work of Robillard et al. [Robillard et al. 2004] to other information-intensive activities, such as code reuse. Moreover, it could develop systems and algorithms for synthesizing sources of data to answer those questions.

5.2 Opportunity: Decreasing cost *C* by decreasing the cost of processing a patch

Many SE tools attempt to reduce the cost of navigating to a patch, commonly by applying the Gather Together pattern; however, only a few tools have attempted to decrease the cost of processing a patch after a developer navigates to it by attempting to help developers to extract information from a method, file, or other patch more efficiently. For instance, syntax-highlighting code editors, no doubt, decrease the cost of reading and understanding a chunk of code. But approaches that are more closely geared to the predator's specific goal have remained nearly untouched.

One general approach might be to make the information content of a patch more salient, and the uninformative portions of the patch less salient, so that the developer can more easily focus on the more useful parts and ignore the less valuable parts. For example, recent research has shown that cloned code is less likely than uncloned code to be buggy [Rahman et al. 2010]. Thus, during a debugging activity, a tool might grey out cloned code to help developers focus attention on code more likely to contain a bug. Conversely, if empirical studies show that specific code smells are positively correlated with the presence of bugs, then the IDE could bold or otherwise highlight that code when the debugger is active. Similar approaches could be taken for deemphasizing or emphasizing certain parts of patches that are written in natural language. For example, researchers have recently proposed a machine learning based approach for summarizing bug reports [Rastkar et al. 2010]. This approach could be extended and adapted to a broader range of SE artifacts, such as requirements and design documents. Tools based on such an approach could potentially enable developers to obtain most of a document's information without the need to process much of the document. Although several algorithms exist for

summarizing natural language (e.g., see [Mani and Maybury 1999]), we are not aware of any for summarizing code.

In short, this tool would first identify parts of patches that are likely to be informative to the developer in a task, and it would then emphasize those parts at the expense of others. To achieve the first of these steps, it seems likely that the algorithms used to identify informative patches could be adapted to identify *parts* of informative patches. Further work might also be helpful for finding novel approaches for deemphasizing or emphasizing parts of patches (e.g., through code summarization).

5.3 Opportunity: Aligning expected cost E(*C*) with real cost *C*

Although several tools help developers to more accurately assess the value of a patch, thereby aligning E(*V*) and *V*, few tools aim at aligning the expected cost E(*C*) with *C*. For example, although recommendation engines often provide snippets or other cues alongside recommendations in order to indicate why the recommended patches might be relevant to the developer's information needs, these recommendation engines provide no indication of how long it will take to process those patches, how hard those patches will be to understand, or other time costs to a developer. As a case in point, instantiations of the Cue Decoration pattern have been mostly used to date to provide information about value, not about cost.

This stands in contrast to work outside of software engineering. For example, as mentioned in Section 2.3, Vigo et al. created a special Web browser for blind users [Vigo et al. 2009]. This browser annotates each hyperlink with an annotation that indicates the target page's "accessibility score" (measuring how well the page could be read by a screen-reader). In IFT terms, this browser provides a cue indicating an estimate of the cost for processing the distal patch. As predicted, experiments showed that when these cues were provided, users were indeed more likely to accurately assess the cost of visiting distant pages, and to select low-cost pages rather than high-cost pages with the same information content.

This approach could be applied in software engineering, using the Cue Decoration pattern to provide information about cost (and not just value). For example, a tool such as Code Miner that recommends certain components for reuse might not only sort components based on their relevance to the task (value), but it could also annotate recommendations with a measurement of time or rating of hassle by previous developers to reuse the code. (Many search engines already allow developers to rate components, and they display average ratings alongside search results, but the ratings' subjectivity and big-picture focus imply that they conflate a component's functionality—its value—with its cost of reuse.) As another example, bug reports could be automatically annotated with an indication of how long or how hard they are to understand, thus helping to inform developers' estimates E(*C*) of bug reports' processing costs.

This approach first requires automatically estimating the cost of processing a patch, then displaying a cue reflecting that information so that developers can make informed assessments of whether they want to invest in visiting that patch. A key obstacle that research could address is finding effective techniques for estimating the cost of processing different kinds of patches.

5.4 Opportunity: Retargeting previously targeted terms in novel ways

The IFT-related commonalities among tools from different areas of software engineering suggest that a new level of generality in the design of SE tools is possible. This points the way to using an IFT lens to investigate the potential of existing SE tools generalizing beyond the particular software engineering activities for which they were initially envisioned. For example, Team Tracks [DeLine et al.

2005] is based on the empirically substantiated observation that if two sections of code are often visited in succession, then they are probably related to one another. When a developer visits one section of code, Team Tracks uses this prey-recognition criteria to provide links to potentially related code to accelerate debugging, and uses the Cue Decoration pattern to clarify the potential of each, so as to engender scent and to align E(*V*) with *V*. An example opportunity is to investigate whether the Team Tracks approach is equally effective at supporting code reuse (or other software engineering activities), because code segments that are related by navigation might make good candidates to reuse together.

As an example of a cross-pollination opportunity, consider that developers often perform refactorings in sequences [Murphy-Hill et al. 2009]. Perhaps an interactive timeline, like the one used by Whyline to support debugging [Ko 2008], could be used to help a developer step back through, examine, and possibly undo refactorings.

All the tools that we surveyed attempted to approximate the predator's goal *G*, but they used a wide variety of approaches for doing so. Another avenue for future work would be to identify recurring patterns in such approaches. For instance, we observed a number of tools that automate the identification of prey by simulating the prey-recognition criteria used by predators. Many refactoring tools (e.g., Soul [Mens et al. 2003], Crocodile [Simon et al. 2001], and jCOSMO [van Emden and Moonen 2002]) automatically look for the bad smells used by developers to identify refactoring opportunities. Furthermore, some reuse tools (e.g., Care [Caldiera and Basili 1991] and Code Miner [Dunn and Knight 1993]) use software metrics to simulate the criteria that developers use to recognize reusable code. Such patterns for approximating prey could potentially support the design patterns in Section 4. For example, the Gather Together pattern could use this information to display a list of links to prey, and the Cue Decoration pattern could use this information to highlight any existing links that point to prey of interest. In other words, prey-approximation patterns could act as computational workhorses that support other patterns that operate on the user interface.

Finally, we have focused on debugging, refactoring, and reuse, but future research could identify ways to apply IFT-based patterns to other information-intensive activities by software developers. For example, requirements validation includes checking consistency between requirements, which may require foraging for information in large collections of requirements documents. Tools could target the navigation cost (and thus *C* overall) by intelligently providing links between documents; they could target the processing cost in *C* by summarizing documents; they could target the value *V* of a document by enabling developers to automatically or manually annotate documents; and they could align E(*V*) with *V* as well as E(*C*) with *C* by intelligently generating cues that inform developers about the value and cost of documents. Likewise, we expect that similar approaches may also be useful for a broad range areas of software engineering such as architectural design, implementation, testing, reverse engineering, configuration management, and other software development tasks that periodically require information foraging.

6. RELATED WORK

6.1 IFT in software engineering research

Despite the widespread success that IFT has had in the domain of Web foraging, only a handful of researchers have applied IFT to software engineering. One recent study of how developers navigate source code used IFT to interpret results from an empirically based model of program comprehension (rather than to make and confirm predictions) [Ko et al. 2006]. A second formative study mentioned that opportunistic developers appear to look for documentation in a manner consistent with IFT

[Brandt et al. 2008] but did not mention how any specific IFT constructs, such as scent, matched up with empirical observations.

To our knowledge, the only *predictive* investigation of IFT in software engineering has been our own, where we operationalized IFT to produce models that quantitatively predicted the navigation behavior of developers during debugging tasks [Lawrance et al. 2008a; Lawrance et al. 2011; Lawrance et al. 2010]. Experiments showed that our operationalizations of IFT could accurately make a variety of predictions about where developers would or should go during maintenance tasks [Lawrance et al. 2008a; Lawrance et al. 2008b; Lawrance et al. 2011; Lawrance et al. 2010]. For example, in a 7-month field study logging the navigation behavior of two professional developers with their normal work in Eclipse, 27% of the navigations were our model's top-ranked prediction, and 50% were within the model's top three predictions [Lawrance et al. 2010].

The bottom line is that IFT has a long and distinguished history of making accurate predictions and guiding interaction designers in the Web domain, but prior to this work, its applicability to software engineering has been largely unexplored.

6.2 Questions that developers ask

Researchers have provided a few alternate conceptual structures aimed at describing specific aspects or kinds of information-intensive activities. Thus, they are more detailed but potentially less general than IFT. For example, following up on an earlier study showing that methodical investigation leads to more effective software maintenance [Robillard et al. 2004], Sillito et al. conducted two think-aloud studies to investigate what information developers seek during such investigations. They identified four categories of questions that developers asked during maintenance tasks [Sillito et al. 2008]:

- Questions about "finding points in the code that were relevant to the task": e.g., "Is there an entity named something like this?"
- Questions "about expanding a given entity believed to be related to the task, often by exploring relationships": e.g., "What data is being modified by this code?"
- Questions "about building an understanding of concepts in the code that involved multiple relationships and entities": e.g., "What is the behavior that these types provide together?"
- Questions "over related groups of subgraphs": e.g., "What are the differences between these files or types?"

Similar empirical studies have uncovered comparable questions asked and information sought by developers (e.g., [Ko et al. 2007; LaToza et al. 2006; LaToza et al. 2007; LaToza and Myers 2010; Layman 2009]).

Questions like the four above can be viewed as instantiations of more abstract questions that an information predator might want to answer:

- "Which patches contain my prey, i.e., the objects with strong scent relevant to my task?"
- "Now that I have found one prey, what nearby patches might have additional prey?"
- "What are the different kinds of prey, and how are they related to one another?"
- "Given the choice between two clusters of patches, which is a better investment of effort, i.e., how do they differ with respect to my present information goal?"

Thus, IFT provides a theoretical complement to a concrete taxonomy situated in a domain, such as Sillito et al.'s. The taxonomy's points have the advantage of

providing a concrete feel for a real developer's specific problems. The theoretical grounding provided by IFT adds to this in three ways.

First, expressing these questions abstractly in terms of IFT helps to explain *why* people would be asking the questions. For example, the Web navigation experiments cited above showed that people often judge scent based on textual similarity, which explains why they would ask for an "entity named something like this" (and other questions in the first category). IFT also predicts that people generally explore each patch (hence questions in the second category) and then make decisions about when and how to navigate to other patches (hence questions in the third and fourth categories).

Second, beyond the questions above that reflect the key constructs of prey, scent, patches and topology, IFT helps to remind researchers not to overlook the other IFT constructs. For example, it predicts that predators will complete tasks faster if they can efficiently *enrich* the environment. This suggests that helping developers not only requires answering questions, but also entails helping them to perform enrichment activities, such as saving search queries, tagging code and other artifacts, or taking electronic notes.

Finally, the abstract IFT viewpoint has several practical advantages for SE researchers. The convergence between the abstract IFT view and concrete views like Sillito et al.'s results lends support to both the validity of the experimental results and the applicability of IFT in this domain. It suggests new generalization opportunities for SE tools from one context to other contexts, such as those described in Section 5. In essence, IFT offers a means for recognizing intellectual connections between research that may not appear to be related without the lens of IFT, potentially enhancing researchers' ability to build on each others' works in a more crosscutting way.

6.3 Recommendation systems for software engineering

In general, recommendation systems provide a form of information filtering that "exploits past behaviours and user similarities to generate a list of information items that is personally tailored to an end-user's preferences" [ACM Recommender Systems 2012]. In the domain of software engineering, recommendation systems aim to provide "information items estimated to be valuable for a software engineering task in a given context" [Robillard et al. 2010]. A qualitative analysis of recommendation systems for software engineering (RSSEs) revealed that they vary along three dimensions [Robillard et al. 2010]. The first dimension is the system's input, whether an explicitly user-specified search context, an implicitly inferred search context, or a hybrid of the two. The second dimension is the nature of the recommendation, both in terms of the data or items recommended (such as bug reports, source code files, or change logs) as well as the structure of the recommendations (how and whether items are ranked). The final dimension is the form of the output, which can be pushed to or pulled by the user, and which can be presented in a separate user interface or seamlessly melded into an existing interface. As cross-dimensional concerns, some systems may include explanations of recommendations, and some may accept and incorporate user feedback.

As with the questions of Sillito et al., applying an IFT perspective can help to explain the success of many aspects of RSSEs—for example, why people might want specific kinds of recommendations in the first place, or why the text-based models frequently used in RSSEs might be appropriate (i.e., because of the Web-based experiments described earlier, showing that text-based similarity captures much of how humans perceive scent). Thus, the theory can be used to argue for the validity of specific systems. It can also be used to provide a theoretical basis for RSSE as an overall approach: RSSE research offers specific insights about particular issues (e.g.,

how to recommend specific kinds of data), with IFT offering an abstract view of why those issues are important and how they fit together (e.g., relevance of certain kinds of code or data to a task).

7. CONCLUSION

The power of a theory lies in its ability to *unify* multiple situations under a common abstraction. In this paper, we have demonstrated how Information Foraging Theory can help SE researchers to understand software developers' information-intensive activities from three areas of software engineering and to support those activities. This paper's IFT perspective therefore makes four contributions:

- We operationalized IFT for different types of information-intensive software engineering activities in ways that *explain* reasons behind the empirical successes of a number of disparate SE tools.
- Building upon these operationalizations, we showed that the IFT perspective can unify disparate information-intensive activities from multiple areas of software engineering under *a common theoretical framework*.
- Generalizing upon tool commonalties revealed by our IFT perspective, we identified a set of IFT *design patterns*, enabling software engineering researchers to take practical advantage of the theory.
- We identified *new research opportunities* suggested by our IFT perspective in multiple areas of software engineering.

Beyond IFT's contribution to help SE researchers further their understanding of how to support developers' information-intensive activities, other human-behavioral theories that have become established in other domains may be useful to understand how to help developers collaborate, make critical problem-solving judgments, and perform other vital software engineering activities. Tapping into additional theories would further help to address the concerns expressed by the SEMAT initiative and by Hannay et al.'s systematic literature review [Hannay et al. 2007]. We hope that one final contribution of this paper may be to provide a set of criteria that researchers can follow in considering how to bring such additional theories to the software engineering domain. Specifically, as with IFT, we would hope that such additional relevant theories would provide abstractions that describe developers' activities in ways that explain why a variety of SE tools succeed, would generalize over multiple software engineering research areas, and would uncover new design patterns that enable SE researchers and tool builders to make productive use of the new theory. In the end, human-behavioral theories would then form a solid foundation for speeding the advance of innovative new software engineering research and tools.

Acknowledgements

This work was supported in part by the Air Force Office of Scientific Research FA9550-09-1-0213, by the EUSES Consortium via ITR-0325273, and by an IBM Open Research Collaboration Award.
We would also like to thank the reviewers for their insightful comments that helped us to improve the paper considerably.

REFERENCES

ACM RECOMMENDER SYSTEMS, 2012. The ACM Conference on Recommender Systems. http://recsys.acm.org/

ALEXANDER, C. 1979. *The Timeless Way of Building*. Oxford University Press.

ASHOK, B., JOY, J., LIANG, H., RAJAMANI, S. K., SRINVASA, G., AND VANGALA, V. 2009. DebugAdvisor: A Recommender System for Debugging. In *Proceedings Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 373–382.

ARLOTTA, L., CRESCENZI, V., MECCA, G., AND MERIALDO, P. 2003. Automatic annotation of data extracted from large Web sites. In *Proceedings of the International Workshop on the Web and Databases*. 12–17.

AULA, A., JHAVERI, N., AND KAKI, M. 2005. Information search and re-access strategies of experienced Web users. In *Proceedings of the International Conference on World Wide Web*. 583–592.

BAEZA-YATES, R. AND RIBEIRO-NETO, B. 1999. *Modern Information Retrieval*. Addison Wesley Longman.

BELLON, S., KOSCHKE, R., ANTONIOL, G., AND KRINKE, J. 2007. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* 577–591.

BLOCH, J. 2001. *Effective Java Programming Language Guide*. Sun Microsystems.

BORCHERS, J. 2000. A pattern approach to interaction design. In *Proceedings of the ACM International Conference on Designing Interactive Systems*. 369–378.

BRAGDON, A., REISS, STEVEN P., ZELEZNIK, R., KARUMURI, S., CHEUNG, W., KAPLAN, J., ADEPUTRA, F., AND LAVIOLA JR., J.J. 2010. Code Bubbles: Rethinking the user interface paradigm of integrated development. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 455–464.

BRANDT, J., GUO, P., LEWENSTEIN, J., AND KLEMMER, S. 2008. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the International Workshop on End-User Software Engineering*. 1–5.

BROOKS, R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies 18*, 543–554, 1983.

BUDIU, R., PIROLLI, P., AND HONG, L. 2009. Remembrance of things tagged: How tagging effort affects tag production and human memory. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 615–624.

CALDIERA, G. AND BASILI, V. 1991. Identifying and qualifying reusable software components. *Computer 24*, 2, 61–70.

CHI, E., PIROLLI, P., CHEN, K., AND PITKOW, J. 2001. Using information scent to model user information needs and actions and the Web. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 490–497.

CHI, E., ROSIEN, A., SUPATTANASIRI, G., AND WILLIAMS, A. 2003. The Bloodhound project: Automating discovery of Web usability issues using the InfoScent simulator. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 512–519.

COBLENZ, M. J., KO, A. J., AND MYERS, B. A. 2006. JASPER: An Eclipse Plug-In to Facilitate Software Maintenance Tasks. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*. 65–69.

COTTRELL, R., WALKER, R., AND DENZINGER, J. 2008. Semi-automating small-scale source code reuse via structural correspondence. In *Proceedings of the ACM/IEEE International Symposium on Foundations of Software Engineering*. 214–225.

CUBRANIC, D., MURPHY, G., SINGER, J., AND BOOTH, K. 2005. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng. 31*, 6, 446–465.

DAGENAIS, B. AND ROBILLARD, M. P. 2010. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings ACM SIGSOFT International Symposium on the Foundations of Software Engineering,* 127–136.

DE ALWIS, B. AND MURPHY, G. C. 2008. Answering Conceptual Queries with Ferret. In Proceedings of the ACM/IEEE International Conference on Software Engineering.

DEARDEN, A. AND FINLAY, J. 2006. Pattern languages in HCI: A critical review, *Human-Computer Interaction 21,* 49–102.

DELINE, R., CZERWINSKI, M., AND ROBERTSON, G. 2005. Easing program comprehension by sharing navigation data. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centered Computing*. 241–248.

DUALA-EKOKO, E. AND ROBILLARD, M. 2007. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*. 158–167.

DUCASSE, S., RIEGER, M., AND DEMEYER, S. 1999. A language independent approach for detecting duplicated code. *1999 International Conference on Software Maintenance*. 109–118.

DUDZIAK, T. 2002. *Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code*. PhD thesis, Technical University of Berlin.

DUNN, M. AND KNIGHT, J. 1993. Automating the detection of reusable parts in existing software. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 381–390.

ERL, T. 2007. *SOA Principles of Service Design*. Prentice Hall.

FOWLER, M. AND BECK, K. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

FRITZ, T. AND MURPHY, G. C. 2010. Using Information Fragments to Answer the Questions Developers Ask. In *Proceedings of the ACM/IEEE International Conference on Software Engineering.*

FU, W.-T. AND PIROLLI, P. 2007. SNIF-ACT: A cognitive model of user navigation on the World Wide Web. *Hum.-Comput. Interact.* 22, 355–412.

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

GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. 2002. Architectural mismatch: Why reuse is so hard. *IEEE Softw. 12*, 6, 17–26.

GEIGER, R., FLURI, B., GALL, H., AND PINZGER, M. 2006. Relation of code clones and change couplings. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*. LNCS 3922, 411–425.

HANNAY, J., SJOBERG, D., AND DYBA, T. 2007. A systematic review of theory use in software engineering experiments. *IEEE Trans. Softw. Eng. 33*, 2, 87–107.

HENNINGER, S. 1994. Using iterative refinement to find reusable software. *IEEE Softw. 11*, 5, 48–59.

HILL, E., POLLOCK, L., AND VIJAY-SHANKER, K. 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 232–242.

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

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

HUNDHAUSEN, C. 2005. Using end-user visualization environments to mediate conversations: A "Communicative dimensions" framework. *Journal of Visual Languages and Computing 16*, 3, 153–185.

JEON, J., LAVRENKO, V., AND MANMATHA, R. 2003. Automatic image annotation and retrieval using cross-media relevance models. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 119–126.

KERSTEN, M. AND MURPHY, G. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering*. 1–11.

KO, A. 2008. *Asking and Answering Questions about the Causes of Software Behaviors*. PhD thesis, Carnegie Mellon University.

KO, A., MYERS, B., COBLENZ, M., AND AUNG, H. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng. 32*, 12, 971–987.

KO, A., 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.

LANUBILE, F. AND VISAGGIO, G. 1993. Function recovery based on program slicing. In *Proceedings of the 1993 International Conference on Software Maintenance*. 396–404.

LANUBILE, F., EBERT, C., PRIKLADNICKI, R. AND VIZCAINO, A. 2010. Collaboration Tools for Global Software Engineering. In *IEEE Software*, March-April, 52–55.

LATOZA, T., GARLAN, D., HERBSLEB, J., AND MYERS, B. 2007. Program comprehension as fact finding. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 361–370.

LATOZA, T. AND MYERS, B. 2010. Developers ask reachability questions. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 185–194.

LATOZA, T., VENOLIA, G., AND DELINE, R. 2006. Maintaining mental models: A study of developer work habits. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 492–501.

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

LAWRANCE, J., BELLAMY, R., BURNETT, M., AND RECTOR, K. 2008b. Can information foraging pick the fix?: A field study. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. 57–64.

LAWRANCE, J., BOGART, C., BURNETT, M., BELLAMY, R., RECTOR, K., AND FLEMING, S. 2011. How programmers debug, revisited: An information foraging theory perspective, *IEEE Trans. Software Engineering,* (preprint available at http://doi.ieeecomputersociety.org/10.1109/TSE.2010.111).

LAWRANCE, J., BURNETT, M., BELLAMY, R., AND BOGART, C. 2010. Reactive information foraging for evolving goals. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 25–34.

LAYMAN, L. 2009. *Information Needs of Developers for Program Comprehension during Software Maintenance Tasks*. PhD thesis, North Carolina State University.

MANI, I. AND MAYBURY, M. 1999. *Advances in Automatic Text Summarization*. MIT Press.

MCMILLAN, C., GRECHANIK, M., POSHYVANYK, D., FU, C., AND XIE, Q. To appear. Exemplar: A Source Code Search Engine for Finding Highly Relevant Applications, *IEEE Transactions on Software Engineering (TSE)*.

MENS, T., TOURWE, T., AND MUNOZ, F. 2003. Beyond the refactoring browser: Advanced tool support for software refactoring. *6th International Workshop on Principles of Software Evolution*. 39–44.

MINTO, S. AND MURPHY, G. C. 2007. Recommending Emergent Teams. In *Proceedings 4th International Workshop on Mining Software Repositories,* 5–13.

MOCKUS, A. AND HERBLSEB, J.D. 2002. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering,* 503–512.

MURPHY-HILL, E. AND BLACK, A. 2008. Refactoring tools: Fitness for purpose. *IEEE Softw. 25*, 5, 38–44.

MURPHY-HILL, E., PARNIN, C., AND BLACK, A. 2009. How we refactor, and how we know it. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*. 287–297.

MYERS, D., STOREY, M.-A., SALOIS, M., AND CHARLAND, P. 2010. Utilizing debug information to compact loops in large execution traces. In *Proceedings of the 14th European Conf. on Software Maintenance and Re-engineering*, 41–50.

OLIVERO, F., LANZA, M., D'AMBROS, M., AND ROBBES, R. 2011. Enabling Program Comprehension through an Object-focused Development Environment. In *Proceedings of the IEEE Symposium on Visual Languages and Human-centric Computing*. 127–134.

OLSON, G. AND OLSON, J. 2003. Human-computer interaction: Psychological aspects of the human use of computing. *Annual Review of Psychology 54*, 1, 491–516.

OLSTON, C. AND CHI, E. 2003. ScentTrails: Integrating browsing and searching on the Web. *ACM Trans. Comput.-Hum. Interact. 10*, 3, 177–197.

PANE, J. 2002. *A Programming System for Children that is Designed for Usability*. PhD thesis, Carnegie Mellon University.

PARNIN, C. AND GÖRG, C. 2006. Building usage contexts during program comprehension. In *Proc. 14th IEEE Intl. Conf. Program Comprehension*, 13–22.

PIORKOWSKI, D., FLEMING, S. D., SCAFFIDI, C., JOHN, L., BOGART, C., JOHN, B.E., BURNETT, M., AND BELLAMY, R. 2011. Modeling Programmer Navigation: A head-to-head empirical evaluation of predictive models, In *Proceedings IEEE Symposium on Visual Languages and Human-Centric Computing*, 109–116.

PIROLLI, P. 2007. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press.

PIROLLI, P. AND CARD, S. 1998. Information foraging models of browsers for very large document spaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*. 83–93.

PIROLLI, P. AND CARD, S. 1999. Information foraging. *Psychological Review 106*, 4, 643–675.

PIROLLI, P. AND CARD, S. 1995. Information foraging in information access environments. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 51–58.

PIROLLI, P. AND FU, W. 2003. SNIF-ACT: A model of information foraging on the World Wide Web. In *Proceedings of the 9th International Conference on User Modeling*. LNCS 2702, 45–54.

RAHMAN, F., BIRD, C., AND DEVANBU, P. 2010. Clones: What is that smell? In *Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories*. 72–81.

RASTKAR, S., MURPHY, G. C., AND MURRAY, G. 2010. Summarizing software artifacts: A case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. 505–514.

REISS, S. 2009. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*. 243–253.

ROBILLARD, M., COELHO, W., AND MURPHY, G. 2004. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.* 889–903.

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

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

SIMON, F., STEINBRUCKNER, F., AND LEWERENTZ, C. 2001. Metrics based refactoring. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. 30–38.

SJØBERG, D. I. K., DYBÅ, T., ANDA, B. C. D., AND HANNAY, J. E. 2008. Building Theories in Software Engineering. *Guide to Advanced Empirical Software Engineering*. Springer, 312–336.

STACK EXCHANGE INC, last accessed 2011. Stack Overflow. *http://stackoverflow.com/*

STOREY, M.-A., CHENG, L., SINGER, J., AND MULLER, M. 2007. How programmers can turn comments into waypoints for code navigation. In *Proceedings of the IEEE International Conference on Software Maintenance*. 265–274.

TIDWELL, J. 2006. *Designing Interfaces*. O'Reilly.

TOKUDA, L. AND BATORY, D. 2001. Evolving object-oriented designs with refactorings. *Automated Software Engineering 8*, 1, 89–120.

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

TREUDE, C., BARZILAY, O., AND STOREY, M.-A. 2011. How do Programmers Ask and Answer Questions on the Web? (NIER Track). In *Proceedings ACM/IEEE International Conference on Software Engineering*. 804–807.

TREUDE, C. AND STOREY, M-A. (to appear). Work Item Tagging: Communicating Concerns in Collaborative Software Development. In *IEEE Transactions on Software Engineering*.

TREUDE, C. AND STOREY, M.-A. 2010. Awareness 2.0: Staying Aware of Projects, Developers and Tasks using Dashboards and Feeds. In *Proceedings ACM/IEEE International Conference on Software Engineering,* 365–374.

TREUDE, C. AND STOREY, M.-A. 2011. Effective Communication of Software Development Knowledge Through Community Portals. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE),* 91–101.

VAN EMDEN, E. AND MOONEN, L. 2002. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. 97–108.

VIGO, M., LEPORINI, B., AND PATERNO, F. 2009. Enriching Web information scent for blind users. In *Proceedings of the ACM International Conference on Computers and Accessibility*. 123–130.

WEXELBLAT, A. AND MAES, P. 1999. Footprints: History-rich tools for information foraging. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*. 270–277.

WÜRSCH, M., GHEZZI, G., REIF, G., AND GALL, H. C. 2010. Supporting Developers with Natural Language Queries. In *Proceedings ACM/IEEE International Conference on Software Engineering*. 165–174.

XING, Z. AND STROULIA, E. 2006. Refactoring practice: How it is and how it should be supported: An Eclipse case study. In *Proceedings of the ACM/IEEE International Conference on Software Maintenance*. 458–468.

YE, Y., FISCHER, G., AND REEVES, B. 2000. Integrating active information delivery and reuse repository systems. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 60–68.

YE, Y., YAMAMOTO, Y., AND NAKAKOJI, K. 2007. A Socio-Technical Framework for Supporting Programmers. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 351–360.

ZIMMERMANN, T., WEISGERBER, P., DIEHL, S., AND ZELLER, A. 2004. Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*. 563–572.