Recommendation Delivery: Getting the User Interface Just Right

Emerson Murphy-Hill and Gail C. Murphy

Abstract Generating a useful recommendation is only the first step in creating a recommendation system. For the system to have value, the recommendations must be delivered with a user interface that allows the user to become aware that recommendations are available, to determine if any of the recommendations have value for them and to be able to act upon a recommendation. By synthesizing previous results from general recommendation system research and software engineering recommendation system research, we discuss the factors that affect whether or not a user considers and accepts recommendations generated by a system. These factors include the ease with which a recommendation can be understood and the level of trust a user assigns to a recommendation. In this chapter, we will describe these factors and the opportunities for future research towards helping getting the user interface of a recommendation system just right.

1 Why Does the User Interface Matter?

Recommendation systems for software engineering (RSSEs) can be divided into two parts: the backend that decides what to recommend, and the frontend that delivers the recommendation. In this chapter, we refer to the **developer** for whom a recommendation is aimed as the **user**. *Toolsmiths*, those developers who design and build RSSEs, often focus on the backend, because one clearly has to have something good to recommend before presenting it to the user.

Emerson Murphy-Hill
North Carolina State University, 890 Oval Drive, Raleigh, NC, United States e-mail: emerson@csc.ncsu.edu

Gail C. Murphy
University of British Columbia, 201-2366 Main Mall, Vancouver, B.C., Canada e-mail: murphy@cs.ubc.ca

Less attention is paid to the user interface than the backend. A toolsmith has many options when choosing a user interface for an RSSE. The first iteration of an RSSE is typically what is easiest to implement, so that is what toolsmiths pick for their first implementation of an RSSE as a demonstration of feasibility, yet never get around to improving the user interface. One could make the case that the reason so few RSSEs have been adopted by the software engineering community is because toolsmiths have spent so little time thinking about the user interface. But let us examine the case of one particular recommendation system outside the domain of software engineering, a case that suggests that the user interface does indeed matter.

Consider Clippy, a user interface agent that recommended new tools to Microsoft Word users. Clippy had a reasonably good recommendation algorithm, backed up by significant empirical research [20]. For example, Clippy would recommend that users try Word's letter template, which may save the user a significant amount of time. However, Clippy was often disliked, even hated, by his user base, enough so that it was listed by Time magazine on its 50 worst inventions list [17]. Why? Many retrospectives pin the blame on the user interface, such as Whitworth's indictment that Clippy was not sufficiently polite [52].

Although RSSEs do not have such a famous example, the lesson is clear – the user interface matters. The user interface mechanisms for auto-complete and varible renaming, which are invoked with a simple keystroke in an editor, are examples of successful user interfaces for RSSEs, as evidenced by the fact that most **integrated development environments** provide convenient mechansisms for toolsmiths to implement them. However, this does not necessarily mean that these two mechanisms are the appropriate mechanism for all information presented by RSSEs. For example, we have argued that for smell detectors [31], a user interface based on underlining code that is potentially involved in a smell is inappropriate, because code smells, such as Long Method [15], are not binary, but are instead matters of degree. Furthermore, even for RSSEs with a firmly entrenched user interface, such as code completion that bring up an overlay in the code editor, it is not clear that the current user interfaces are the *best* mechanisms to represent recommended information – perhaps they are simply the mechanisms that people are most accustomed to.

Other communities have likewise realized the importance of user interfaces. For example, collaborative filtering-based recommendation systems, such as GroupLens [22], have seen increasing attention paid to the user interface in recent years. Konstan and Riedl refer to a turning point when it became evident that the evaluation and design of the user experience for a recommendation systems was as important as ensuring the underlying algorithms were accurate [23]. Work on the user experience in the collaborative filtering community focuses on such aspects as **personalization**, ratings and **privacy**. RSSEs typically use algorithms that are less personalized and thus our focus in this chapter is on different factors than are reported on in the collaborative filtering-based recommendation community.

In this chapter, we first discuss several factors that affect a user's likelihood to be receptive to a recommendation. Next, we discuss the space of options a toolsmith has when creating an RSSE user interface, and discuss some of the advantages and disadvantages of those options. Then, we review some techniques toolsmiths can use in the design and evaluation of the user interface of an RSSE.

2 Presenting Recommendations

The user interface of an RSSE must present recommendations in a manner that allows users to consider acting upon the recommendations. Presenting the recommendations requires a user interface of some kind. The toolsmith must make many choices when designing the user interface for a recommender. We describe five factors the toolsmith must consider: understandability, transparency, assessability, trust and timing. As we describe the factors, we provide examples of RSSE user interfaces that have made different choices. We also describe how the factors interact.

2.1 Understandability

Understandability refers to whether a user is able to determine *what* a recommender is suggesting. There are two primary dimensions to understandability: *obviousness* and *cognitive effort*. These two dimensions are independent. A user interface can be non-obvious, but once learned, may require significantly less cognitive effort.

The obviousness dimension describes how easy or hard it is for a user to recognize the kind of recommendation being provided. What a recommendation is may be readily apparent to a user. For instance, a duplicate bug recommender (e.g., [18]) that brings up other bug reports from the same project when a new report is being considered provides a recommendation that is obvious for the user: the recommended bugs are in a style and form that a user immediately recognizes as a bug report. When a recommendation is obvious, little or no training is needed to describe what the recommendation is to a user. At the other end of the scale, a recommender that suggests properties of an artifact may require more training. For example, StenchBlossom displays information about design defects in a users' code by displaying an abstract visualization that the user must explore and interpret in order to comprehend [31].

The effort dimension describes how much cognitive effort is required to uptake the meaning of the recommendation when it is presented. A recommendation for which the cognitive effort to understand is at-a-glance will be easy for a user, once trained, to recognize the meaning. For example, the size of a petal in StenchBlossom maps to "how bad" a problem is, and users can simply glance at the visualization to interpret it, once they have trained themselves to interpret it. As an example at the other end of the scale, if a user is recommended a source code element, such as a method, which may be related to a current element being edited [11], the cognitive effort to understand the meaning of the recommendation may be much higher.

How can a toolsmith improve understandability in their system? Nielsen provides several heuristics for user interface design, three of which are applicable for understandability in RSSEs [34]. The first is “match between system and the real world”, which suggests that the system should use words, phrases, and concepts that the user has likely encountered previously. The second is “consistency and standards”, which suggests that the system should follow conventions and not make the user wonder whether two words or concepts mean the same thing. The third is “help and documentation”, which suggests that the system should be usable without help, but help should be provided when required in a searchable, task-focused, concrete, and minimal manner.

2.2 Transparency

In addition to understanding what a recommendation is, a user must be able to determine *why* the recommendation is being provided. Similar to other earlier works [45], we refer to this factor as *transparency*.

Transparency is related to rationale. If it is clear why a recommendation is being given, the transparency is high. Using our example of a duplicate bug recommender, it may be straightforward to provide transparency if the recommendation is based on similarity of text by reporting a percentage of similarity or by indicating stack traces that match exactly.

When a recommendation is based on more than a simple measure, describing the rationale for the recommendation may be more difficult. For example, a recommender that suggests a likely more efficient command to use in a development environment may require more substantial text or pictures to explain how the new command replaces other commands. For example, the user interface for the Spyglass system [50], which provides command recommendations, shows the user a rationale that describes the intended action, such as intending to navigate a call relationship between two specific points in the code, and the various ways of invoking the more efficient recommended command, such as using a call graph tool through a keyboard shortcut or a menu item.

When transparency is low and rationale is needed, the content and presentation of the rationale can have an effect on the user’s perception of the recommender, such as the user’s trust in the system [47]. The user modeling and collaborative-filtering recommendation communities have performed many studies into the effect of different styles of explanation on user behavior (e.g., [29]).

How can a toolsmith improve transparency in their system? In general, the more information that the sytem can provide about the rationale for a recommendation, the better. This information is generally available to the underlying RSSE algorithm, and transparency is merely a matter of providing it to the user. However, the challenge is doing it in a way that remains understandable. In the field of general recommender systems, Ozok and colleagues advise concrete explanations, such as

"People who use X also use Y", over abstract ones, such as "You may also like Y" [39].

2.3 Assessability

Once a user understands what a recommendation is and why it is being provided, there is still a need to *assess* whether or not a recommendation is relevant and is one that a user wants to take action upon.

Recommendations provided in software development typically require higher assessment than those provided in consumer-oriented domains. For instance, recommending a related news article to a user [22] may be assessable in a split second; does the title of the article appear interesting? In contrast, recommending a potential duplicate bug report requires gaining an understanding of the recommended report and comparing that against the new report: a cognitively challenging task. If seven potential duplicates are presented to the user, at least six comparisons are needed with substantial cognitive shifting between each comparison.

There is a spectrum of assessability in software engineering recommenders. At one end of the spectrum, it may be relatively easy for a user of Reverb [43], which recommends a web page the user has previously visited relevant to code currently being edited, to determine if the web page is of use for the current task. The assessment in this case may be simple because the user may recall the web page from previous interactions. At the other end of the spectrum, it may be difficult for a user of Fishtail [42], which recommends an likely relevant but not necessarily previously visited web page, to determine if the web page is useful as it may take them significant time and effort to read through the recommended page. In general, the longer it takes a user to assess a recommendation, the higher the cost of **false positives**false positive and the more a recommender needs to be accurate.

Assessability is related to, but different from, understandability and transparency. Easy to use and transparent recommendations may be more likely to be easy to assess. A recommendation of a web page that a user has previously visited when they previously edited code may be easy to understand, transparent and easy to assess. However, a recommendation can be easy to understand and transparent yet hard to assess. The difficulty of assessing the duplicate bugs outlined above is an example of this case. More difficult to understand and less transparent recommendations may be acceptable if once the recommendations are assessed they are almost always applicable. For instance, a highly accurate duplicate bug recommender may be acceptable and considered useful and efficient to a user.

How can a toolsmith improve assessability in their system? If a recommendation is an alternative to what the user already has already done (such as recommending a duplicate bug report), the system should make it easy for the user to compare the existing item and the recommended item. In the duplicate bug report example, the system can highlight the salient differences between bug reports. If multiple comparisons are necessary, a higher level difference summary may be appropriate. In general, the system can make it easy for the user to assess the value of a recommendation by comparing the recommendation against the alternatives with respect to the user's values.

2.4 Trust

Even if a recommendation is understandable, transparent, and assessable, a users must **trust** the recommendation to benefit them in the way the recommender system implies. The need to establish trust varies with the level of commitment that the user needs to make with using a recommender. At one end of the spectrum, an RSSE that makes a recommendation to change a user's code and that will make the change automatically requires a significant amount of trust. If the automatic change were to introduce a subtle bug, the user may not recognize it until long into the future, and it may take a significant amount of time to track down and fix. On the other hand, an RSSE that auto-completes a method name may not require much trust from a user, because if the user does not like the chosen method, they simply delete the identifier immediately.

Trust is especially important in RSSEs that necessitate behavior changes on the part of the user. For example, we have previously created an RSSE that recommends integrated development environment tools to users [50]. Such RSSEs might recommend, for example, that a user use a "Call Hierarchy" tool in an integrated development environment such as Eclipse [7], rather than using multiple invocations of a "Find References" command. Even though the RSSE provided high levels of transparency, which can help build trust [40], users found the recommendations hard to trust.

So, how can a toolsmith enable the user to trust their system?

- **Build it.** Start with a small, modest recommendation before making more substantial recommendations. Although, to our knowledge, user interfaces in RSSEs have not allowed users to provide explicit feedback, such feedback has been shown to increase trust in other recommender systems [33].
- **Borrow it.** Borrow trust from someone or something that already has it, such as a colleague of the user. For example, rather than saying "when making this change, you could also look at class X," instead say that "when making similar changes, your colleague Bill often looks at class X."
- **Fake it.** Because humans are social creatures, they are influenced by social cues, cues that could be leveraged to improve recommendation acceptance. Cialdini gives six principles of persuasion that could be leveraged in RSSEs: reciprocity, commitment and consistency, social proof, authority, liking, and scarcity [5]. For example, a toolsmith could use authority in the RSSE by appealing to the fact that the recommendation is derived from Ph.D. work that has analyzed millions of lines of source code. Cialdini's principles have been used successfully outside of software engineering to improve recommendations [10].

2.5 Distraction

When should an RSSE make a recommendation? Many types of RSSEs make recommendations when the user explicitly asks for it; in these cases, the answer is clear – deliver the recommendation when it is asked for. For RSSEs where the tool needs to take the initiative, the answer is less clear.

For some RSSEs, delivering a recommendation in the middle of a user's work is critical. For example, BeneFactor can suggest that a user complete a manual **refactoring** using a refactoring tool [16]. In this case, the longer the RSSE waits to make the recommendation, the less time the user will save in taking the recommendation. If the user completes the refactoring manually, the recommendation has lost its value.

There are downsides to delivering early (and potentially frequent) recommendations as delivering a recommendation in the middle of a user's task may be distracting. That is, the cost of the interruption may outweigh the benefit that the recommendation brings, assuming the user even realizes that benefit.

How can a toolsmith reduce distraction in their system? Several user interface techniques have been proposed to help balance the need for timely recommendations with the need to avoid distracting users. One is the use of negotiated interruption [28], which informs the user that a recommendation is available without forcing the user to acknowledge it immediately. Annotations (Section 3.1) are one implementation of negotiated interruptions – the user can easily ignore or defer the recommendations that these affordances contain.[1] Another is the use of attention sensitive alerting [20], where the recommender system tries to infer when the user is not in the middle of an important task. Carter and Dewan have created such a system that gives help to developers when it detects that they are stuck [4]. Adamczyk and Bailey provide a good overview of techniques designed to reduce distraction in general human computer interaction that can be applied to RSSEs [1].

3 Strategies Used in RSSE User Interfaces

Many existing ways to present recommendations exist. We divide this presentation into two pieces: getting the user's attention (Section 3.1), and providing further information (Section 3.2).

[1] We use the term human-computer interaction term "affordance" to refer to "the actionable properties between the world and an actor" [38].

3.1 Interfaces for Getting Users' Attention

As we hinted at earlier, how contact is initially made between a user and an RSSE is one major user interface decision when designing an RSSE. One of the major distinctions is **reactive** versus **proactive** initiation [54]. Reactive initiation means that when users are ready to receive a recommendation, they ask the tool for it. Proactive initiation does not require the user's invocation; instead, the tool makes a recommendation when it is programmed to do so, perhaps at a scheduled time or perhaps because of some event. Schafer and colleagues' classic paper on e-commerce recommenders calls this distinction "automatic" versus "manual" recommendations [44]. Elsewhere, systems implemented with proactive initiation have also been called "active help systems" [12].

Not all user interfaces fit cleanly into these two categories. For example, Quick Fix Scout piggybacks quick-fix recommendations on top of an existing recommender system user interface [30]; the user does not have to explicitly ask for a recommendation from Quick Fix Scout, but neither is one offered at a particular time.

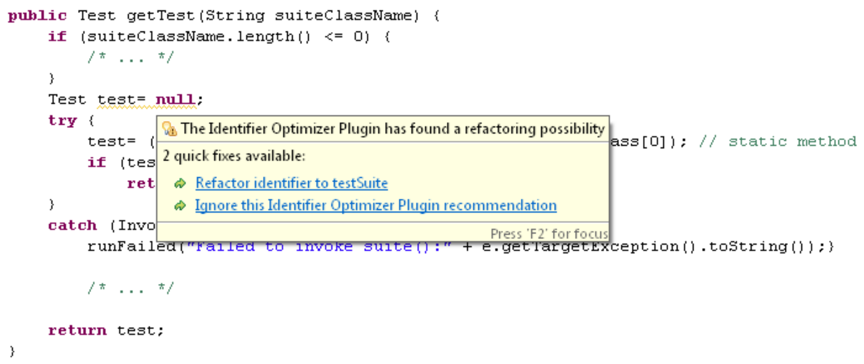
Proactive recommendations tend to be appropriate whenever a recommendation is timely, that is, it may significantly improve the task that the user is doing at the time the recommendation is made. Reactive recommendations tend to be appropriate when the delivery of a recommendation does not impact a particularly time sensitive task, and when communication of a recommendation takes a significant amount of time.

Reactive recommendations tend to be easier to implement; the toolsmith provides a button or hotkey, and users invoke it when they want a recommendation. Finding such a button or a hotkey can be a challenge for the user [32]. There is a significant challenge, however, in designing and implementing a proactive initiation recommendation system. In the subsections in this section, we discuss several existing user interfaces for facilitating proactive initiation.

Annotations Annotations are markup on program text that associate a particular recommendation with the segment of text that they are displayed on. Annotations are often represented as squiggly underlines or highlights. Figure 1 shows an example built by Thies and Roth, where a yellow underlining of the code `test= null` is augmented by a text hover, providing additional information [46].[2]

The advantage of annotations is that they are often familiar interfaces for software developers, and many integrated development environments make it easy to implement them. They also appear in a convenient location whenever a recommendation is associated with a specific code location, so that when developers look at the code, they are likely to notice the recommendation. For example, in Figure 1, the annotations work well because the recommendation speaks to the variable declaration on which the annotation is displayed.

[2] All screenshot images used in this chapter have been provided by their respective authors, each of which has licensed the image under Creative Commons Attribution license (`http://creativecommons.org/licenses/by/3.0/`).



```
public Test getTest(String suiteClassName) {
    if (suiteClassName.length() <= 0) {
        /* ... */
    }
    Test test= null;
    try {
        test= (                                        ass[0]); // static method
        if (tes
            ret
    }
    catch (Invo
        runFailed("Failed to invoke suite():" + e.getTargetException().toString());}

        /* ... */

    return test;
}
```

The Identifier Optimizer Plugin has found a refactoring possibility

2 quick fixes available:

- Refactor identifier to testSuite
- Ignore this Identifier Optimizer Plugin recommendation

Press 'F2' for focus

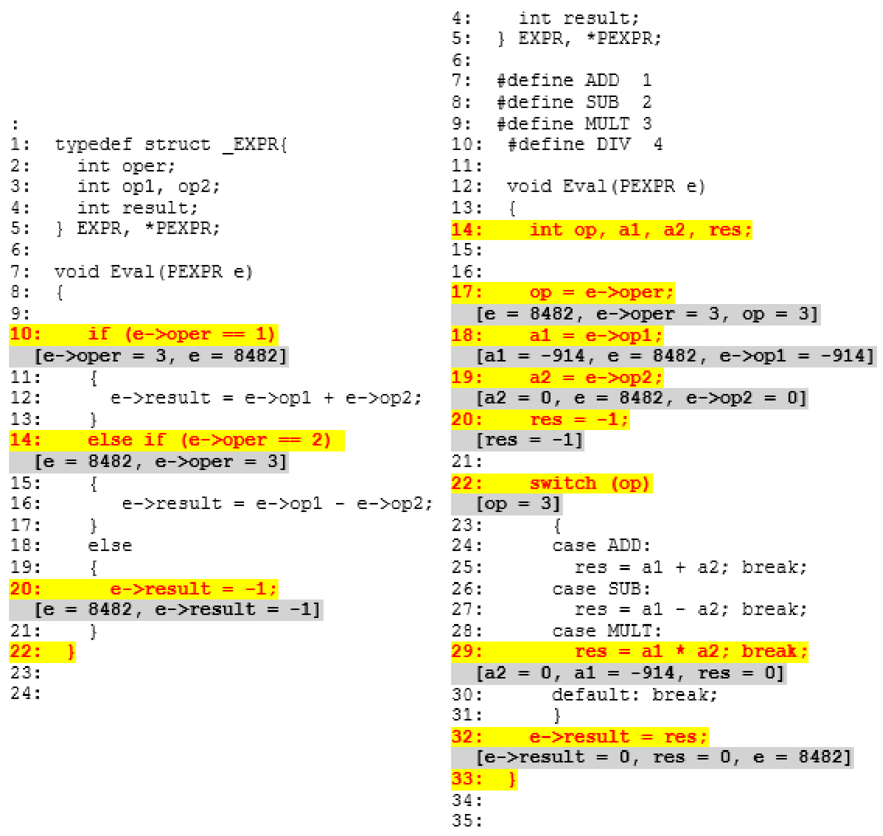
Fig. 1: An example of an annotation, courtesy of Thies and Roth [46]. Here, the an information popup is shown after the mouse has hovered over the annotation.

However, annotations are not well-suited for some situations: when recommendations occur *frequently* in source code, are *soft*, are *imprecise*, or *overlap*. Frequent annotations are those that would be scattered all over the code, overloading the user to the point of ignoring the recommendations or turning them off. For example, Fowler suggests that comments are indicators of poor design [15], but an RSSE that annotated all comments would be excessive. Soft recommendations are those that require human judgment, such as what it means for a method to be "too long" [15]. Imprecise recommendations are those that could reasonably be placed on multiple points in code; for example, a recommendation that coupling should be reduced between classes could as easily be annotated on the referencing class as the referenced class. Overlapping recommendations are those that would overlap if the source code were annotated; for example, if multiple tools all annotated the same expression, at a glance the developer would not be able to distinguish one annotation from multiple annotations.

A special type of annotations are what might be called "document splits," where information is inserted between lines in a document. Figure 2 shows an example, where line numbers are shown along the left hand side; between some lines, information about variable values are displayed. Document splits can display more information initially than other kinds of annotations, but also may be significantly more distracting because they distort a user's documents.

Icons Icons are small graphic images that appear in a development environment. Icons are typically displayed on the periphery of the user's workspace, sometimes as markers in the gutter of an editor. Figure 3 shows an example of the BeneFactor tool [16] recommending that the developer should use a refactoring tool.

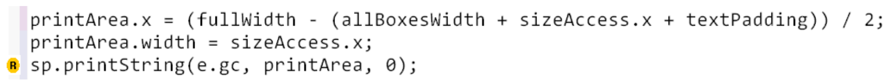
Icons share many of the advantages and disadvantages of annotations, but because icons do not occupy the same screen space as code, icons may be less noticeable than annotations when the user does not happen to glance in the direction of the icon.



```
typedef struct _EXPR{
  int oper;
  int op1, op2;
  int result;
} EXPR, *PEXPR;

void Eval(PEXPR e)
{

   if (e->oper == 1)
 [e->oper = 3, e = 8482]
   {
     e->result = e->op1 + e->op2;
   }
   else if (e->oper == 2)
 [e = 8482, e->oper = 3]
   {
      e->result = e->op1 - e->op2;
   }
   else
   {
     e->result = -1;
 [e = 8482, e->result = -1]
   }
}
```

```
  int result;
} EXPR, *PEXPR;

#define ADD  1
#define SUB  2
#define MULT 3
 #define DIV  4

 void Eval(PEXPR e)
 {
   int op, a1, a2, res;


   op = e->oper;
 [e = 8482, e->oper = 3, op = 3]
   a1 = e->op1;
 [a1 = -914, e = 8482, e->op1 = -914]
   a2 = e->op2;
 [a2 = 0, e = 8482, e->op2 = 0]
   res = -1;
 [res = -1]

   switch (op)
 [op = 3]
     {
     case ADD:
       res = a1 + a2; break;
     case SUB:
       res = a1 - a2; break;
     case MULT:
       res = a1 * a2; break;
 [a2 = 0, a1 = -914, res = 0]
     default: break;
     }
   e->result = res;
 [e->result = 0, res = 0, e = 8482]
 }
```

Fig. 2: An example of a document split from the SymDiff tool, courtesy of Lahiri [24].



```
printArea.x = (fullWidth - (allBoxesWidth + sizeAccess.x + textPadding)) / 2;
printArea.width = sizeAccess.x;
sp.printString(e.gc, printArea, 0);
```

Fig. 3: An example of an icon (at left) from the BeneFactor tool, courtesy of Ge [16].

Affordance Overlays Affordance overlays are annotations that appear on top of user interface affordances, such as files in a browser or items in a dropdown menu. For example, Figure 4 shows a set of task contexts, where one source code file is overlaid with a rounded rectangle, for the purpose of offering the user a recommendation that this is the file he should look at next.

Affordance overlays are well suited to recommendation contexts where a recommendation is frequent or constant and where the existing development environment's user interface should be as unperturbed as possible. Affordance overlays

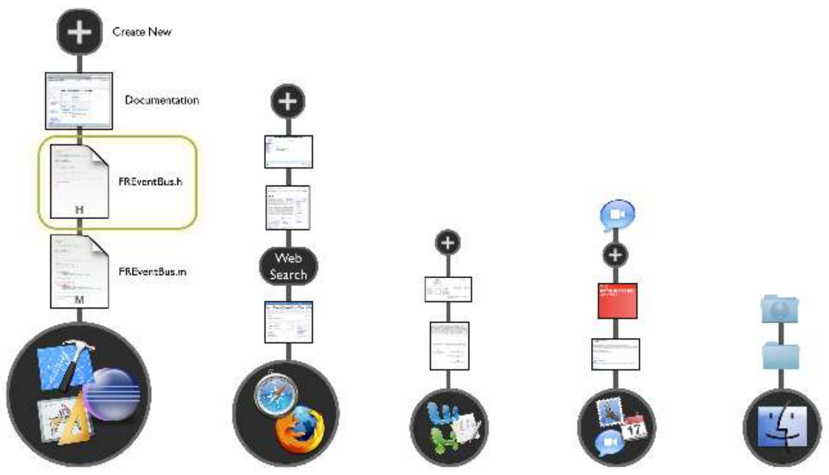


Fig. 4: An example of an overlay from the Switch! user interface, courtesy of Maalej [26].

may not work well in high-stakes situations, when a user missing a recommendation would have a significant impact.

Popup Popup (or toaster) recommendations are those that appear in a new user interface layer when a recommendation is made, on top of an existing user interface. Popups may force the user to acknowledge them, or may disappear after some amount of time. Figure 5 shows a basic popup that Carter and Dewan used for helping software developers when they get stuck [4]. Popups typically disappear after a few seconds, but also have various degrees of ephemerality, from disappearing completely to leaving behind a affordance (such as an icon) that the user can invoke the retrieve the recommendation after the popup itself has disappeared.

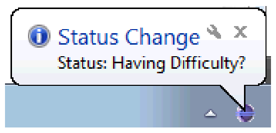
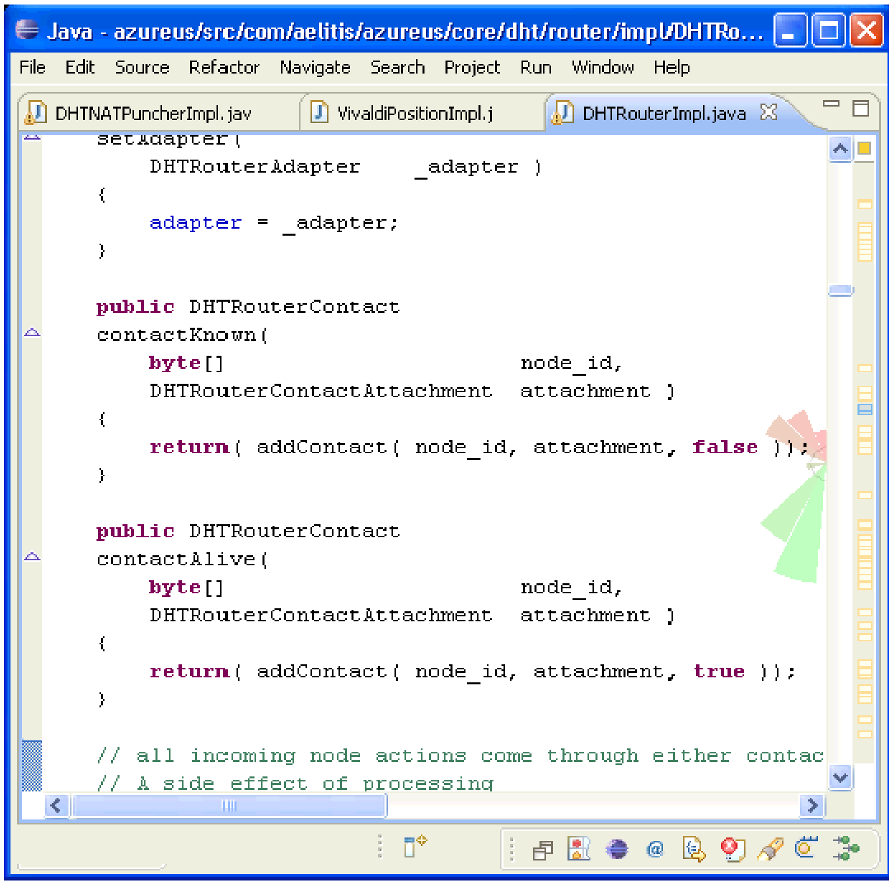

Status Change
Status: Having Difficulty?

Fig. 5: An example of a popup, courtesy of Carter and Dewan [4].

Popups may work well in situations where getting the user's attention to deliver a recommendation is a high priority and when recommendations are infrequent. Popups may not work well when the likelihood of the user taking the recommendation is low.

Dashboard A dashboard is a user interface affordance where recommendations are made to users in a fixed, known location on the screen, typically on the periphery of the user's vision, allowing the user to glance at recommendations frequently and with low commitment. Like the dashboard on a car, recommendation system dashboards typically integrate recommendations of different types or from different sources. Figure 6 shows an example of a dashboard, StenchBlossom, which continuously displays information about multiple code smells while the developer works.


Java - azureus/src/com/aelitis/azureus/core/dht/router/impl/DHTRo...
File Edit Source Refactor Navigate Search Project Run Window Help
DHTNATPuncherImpl.jav
VivaldiPositionImpl.j
DHTRouterImpl.java

```
setAdapter(
    DHTRouterAdapter     _adapter )
{
    adapter = _adapter;
}

public DHTRouterContact
contactKnown(
    byte[]                        node_id,
    DHTRouterContactAttachment    attachment )
{
    return( addContact( node_id, attachment, false ));
}

public DHTRouterContact
contactAlive(
    byte[]                        node_id,
    DHTRouterContactAttachment    attachment )
{
    return( addContact( node_id, attachment, true ));
}

// all incoming node actions come through either contac
// A side effect of processing
```

Fig. 6: An example of a visualization of code smells, where each 'petal' represents the magnitude of a single code smell in the code on the screen [31]. For example, the bottom-most petal indicates a strong Large Class smell [15].

Dashboards may work well in situations where recommendations are continuous and pervasive. They may not work well when a user does not have the screen real estate to spare to the dashboard.

Email Notification Email notifications are those that are delivered via email, rather than into an integrated development environment. One example is email notifications delivered by the Coverity static analysis tool [9], which can notify developers of potential defects via email.

Emails notifications may work well in situations where every recommendation should be considered by a user, and in situations where collaborating with outside entities (such as project managers) may be essential when a developer deals with a recommendation. Because users may be notified about new emails according to their email client preferences, email notifications of recommendations is a way to provide the user with enhanced customizability and workflows. Since email is asynchonous, email may not work well in situations where recommendations must be handled immediately.

3.2 Descriptive User Interface Options

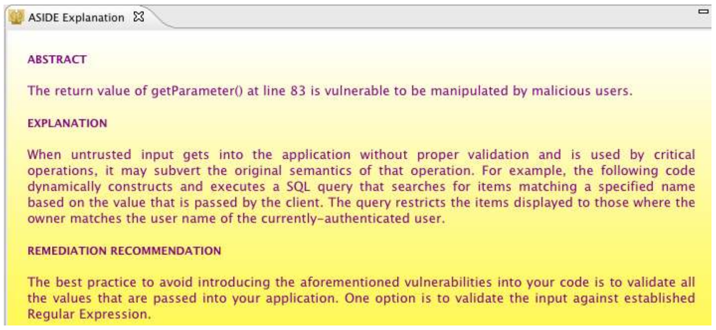
Beyond making initial contact with a user, toolsmiths often want their recommendation systems to provide additional information to a user. Since providing such information in the initial contact may be overwhelming, toolsmiths can employ progressive disclosure [35] to give the user more information. As we discuss in the following subsections, broadly speaking, this information can be conveyed in a textual way, as a transformation, or as a visualization. These user interface options for providing recommendation descriptions are equally appropriate for both proactive and reactive recommender systems.

Textual A textual description is one that explains a recommendation in text. Textual descriptions can be enhanced by using markup, visual emphasis, and a tabular format. Figure 7 gives an example of a textual description from the ASIDE security tool, which explains why a developer should fix an input vulnerability. Many other tools provide textual descriptions, including Niu and colleagues' tool for recommending conflict resolution [37], the Example Overflow tool that recommends relevant source code [56], and the Seahawk tool that recommends relevant answers from a question and answer site [2].

Textual descriptions may be appropriate when a recommendation requires significant context and rationale. However, textual descriptions may not be appropriate when users have little time to read the text.

Transformative Transformative recommendations are those that show the user the impact of taking a recommendation. The impact might be what happens when a tool is invoked or when code is changed. Figure 8 shows an example where a refactoring is recommended. Before this screenshot was taken, the developer had cut the code

```
int string_size=string.length();
size+=string_size;
```



ABSTRACT

The return value of getParameter() at line 83 is vulnerable to be manipulated by malicious users.

EXPLANATION

When untrusted input gets into the application without proper validation and is used by critical operations, it may subvert the original semantics of that operation. For example, the following code dynamically constructs and executes a SQL query that searches for items matching a specified name based on the value that is passed by the client. The query restricts the items displayed to those where the owner matches the user name of the currently-authenticated user.

REMEDIATION RECOMMENDATION

The best practice to avoid introducing the aforementioned vulnerabilities into your code is to validate all the values that are passed into your application. One option is to validate the input against established Regular Expression.

Fig. 7: An example of a textual description from the ASIDE tool, courtesy of Xie, Lipford, and Chu [55].

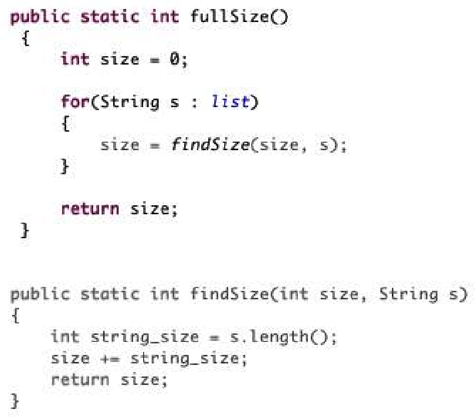
out of the `for loop`, and has begun typing `findSize` into that loop. The recommendation system, WitchDoctor, then recommends that the developer create a new method with the cut code using the appropriate parameters and return value. The recommendation is made in a transformative way, because the gray code (for example, "`size =`", "`(size, string);`", and so on) previews what would happen if the user accepts the recommendation. A more conventional implementation of transformative recommendations might simply show a preview of a change in a separate windows or popup. ChangeCommander is an example of a recommendation system that takes this approach [13]. A similar approach would be to allow the recommender system to make a code change, but then allow the developer to undo that change.

Transformative recommendations may work well in situations where the consequence of a recommendation is known. They may not work so well in situations where the consequence takes a significant amount of time for the user to understand, since the user must essentially reverse engineer the transformation to understand the problem that it solved.

Visualization. Visualizations convey recommendations in a graphical way. Figure 9 shows a visualization of a callers and callees in a piece of software. Trumper and Dollner provide an overview of visualization techniques for RSSEs [49].

Visualizations may work well when recommendations are indirect and require a software developer's judgment. In essence, visualizations collect and display information, with the hope that the developer will take action based on the information that the RSSE provides. This is contrast to many textual recommendations, which precisely tell the developer what to do.

In this section, we have discussed several user interfaces that toolsmiths can use to present their recommendations. However, we do not want the reader to treat this list as exhaustive – our opinion is that novel user interfaces may better fit the needs



```
public static int fullSize()
 {
     int size = 0;

     for(String s : list)
     {
         size = findSize(size, s);
     }

     return size;
 }


public static int findSize(int size, String s)
{
    int string_size = s.length();
    size += string_size;
    return size;
}
```

Fig. 8: An example of a transformative recommendation from the WitchDoctor tool, courtesy of Foster and colleagues [14].

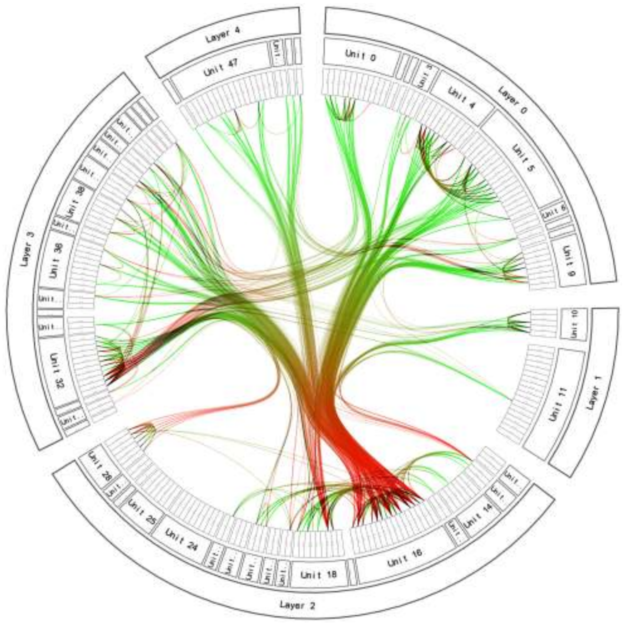


Fig. 9: An example of a visualization from a call tree, courtesy of Holten [19].

of users. Such novel interfaces may completely discard our list of user interfaces, or combine them in a novel way.

The user interfaces presented in this section may seem to be intuitively related to the dimensions we describe in the prior section, but we view the two as orthogonal issues. For example, one might assume that a popup user interface is more distracting than an affordance overlay. However, the fact that many popups appear distracting is inessential to that user interface; a toolsmith can reduce the distraction of popups by using techniques such as gradual fading and more intelligent popup timing.

4 Choosing the Right User Interface

In this chapter we have provided a review of some desirable properties of RSSEs and a variety of user interfaces to help implement those properties. When toolsmiths have an idea for a user interface, how do they design the user interface that is going to deliver recommendations in an effective way? Here, we provide some practical advice on how to do so.

First, toolsmiths should determine their level of commitment to getting the user interface right. At one end of the spectrum, if the toolsmith were just doing a proof of concept, it may be enough to make the most convenient user interface – or not even have a user interface at all. For example, WitchDoctor [14] recommended refactorings, but it was evaluated without actually showing the RSSE to developers, and thus a user interface was not necessary to demonstrate effectiveness. At the other end of the spectrum, if a toolsmith wants the tool to be widely adopted by the user community, we suggest making a stronger commitment to implementing a user interface right.

Second, a toolsmith should choose a strategy for creating a good user interface that is congruent with the level of commitment. Typically such a strategy involves two symbiotic parts, design (creating the user interface) and **evaluation** (determining the goodness of the user interface). In the following paragraphs, we describe several strategies for performing design and evaluation, both with low levels of commitment and with high levels of commitment.

With a low level of commitment, here are a few appropriate design strategies:

- **Use another interface for inspiration**. If a user interface was designed to solve one problem, a tool that needs to solve a similar problem may be designed with a similar interface. For example, because both compiler warnings and static analysis warnings serve similar purposes, annotations that are used for compiler warnings may be appropriate for static analysis warnings.
- **Mockups**. Mockups allow an RSSE designer to create an initial idea for an RSSE user interface, then communicate that idea graphically. A mockup may be created using Powerpoint [8], Adobe Fireworks [6], or simply on paper. For example, the authors of WitchDoctor provided a user interface mockup in their paper, to give the reader an idea of what a practical implementation might look like [14].

- **Cognitive Walkthroughs**. Starting with a basic user interface design (say, a mockup), a toolsmith can then "walk through" how a user would use it for the first time by creating a number of scenarios. By pretending to use the design for each scenario, a toolsmith can determine which scenarios the user interface appears to work well, Rieman and colleagues provide an overview of the cognitive walkthrough procedure [51].

With a low level of commitment, here are a few appropriate evaluation strategies:

- **Wizard Of Oz**. This technique provides a way for toolsmiths to evaluate the user interface of RSSEs without implementing the RSSE fully; instead, the toolsmith manually provides fake (but useful) recommendations directly through the user interface. Maulsby and colleagues provide an introduction to the approach [27].
- **Heuristic Evaluation**. This technique is a way to evaluate RSSE user interfaces by having a panel of experts analyze a user interface by comparing its features to a set of known good **usability** heuristics. Nielsen and Molich provide an overview [36].

With a high level of commitment, appropriate design strategies include those that fall under the heading of **requirements** elication and analysis. Indeed, a toolsmith can treat RSSEs like any piece of software, and design RSSEs using methodologies such as Participatory Design or Joint Application Design [3].

Evaluation strategies with high levels of commitment include:

- **A/B testing**. This type of evaluation, more commonly found in web design, gives one sample of users one user interface, and another sample of users a slightly different user interface [21]. Across both samples, a toolsmith measures an outcome of interest, such as the number of recommendations taken.
- **Controlled experiments**. In controlled experiments, different user interfaces to RSSEs are given to different groups of people and some outcome is measured, but external variables (such as task) are controlled for [53]. As with A/B testing, a toolsmith measures some outcome of interest, and results are compared between groups. However, non-comparative controlled experiments can also be conducted when no reasonable point of comparison exists.
- **Case studies**. Case studies are like controlled experiments, except they are not conducted in controlled conditions, but instead are conducted in a user's usual workspace, which improves their **generalizability** [41]. Data can be collected through a number of means, such as remote instrumentation or by asking users to keep journals.

In addition to the reference given above, Toleman and Welsh provide a more in-depth overview of evaluating design choices [48]. LaToza and Myers provide an overview of the software engineering design process [25], much of which is applicable to RSSEs. Chapter **??** also discusses an end-to-end design and evaluation approach called a **field study**.

Getting the user interface right takes time and effort, but it is also a necessary step in creating a successful recommendion system for software engineering. In this chapter, we have outlined factors that a toolsmith should consider when building a user interface for an RSSE, provided examples of choices the toolsmith can make and have described how user interfaces for RSSEs can be progressively designed and evaluated.

References

1. P. D. Adamczyk and B. P. Bailey. If not now, when?: the effects of interruption at different moments within task execution. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 271–278, New York, NY, USA, 2004. ACM.
2. A. Bacchelli, L. Ponzanelli, and M. Lanza. Harnessing stack overflow for the ide. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 26 –30, june 2012.
3. E. Carmel, R. Whitaker, and J. George. Pd and joint application design: a transatlantic comparison. *Communications of the ACM*, 36(6):40–48, 1993.
4. J. Carter and P. Dewan. Design, implementation, and evaluation of an approach for determining when programmers are having difficulty. In *Proceedings of the 16th ACM international conference on Supporting group work*, GROUP '10, pages 215–224, New York, NY, USA, 2010. ACM.
5. R. Cialdini. *Influence: Science and practice*, volume 4. Allyn and Bacon Boston, MA, 2001.
6. Adobe fireworks. `http://www.adobe.com/fireworks`, Jan. 2013.
7. Eclipse. `http://www.eclipse.org`, Jan. 2013.
8. Microsoft powerpoint. `http://microsoft.com/powerpoint`, Jan. 2013.
9. Coverity. Effective management of static analysis vulnerabilities and defects, 2011. `http://www.coverity.com/library/pdf/effective-management-of-static-analysis-vulnerabilities-and-defects.pdf`.
10. P. Cremonesi, F. Garzotto, and R. Turrin. Investigating the persuasion potential of recommender systems from a quality perspective: An empirical study. *ACM Trans. Interact. Intell. Syst.*, 2(2):11:1–11:41, June 2012.
11. R. DeLine, M. Czerwinski, and G. G. Robertson. Easing program comprehension by sharing navigation data. In *VL/HCC*, pages 241–248. IEEE Computer Society, 2005.
12. G. Fischer, A. Lemke, and T. Schwab. Active help systems. *Readings on Cognitive Ergonomics – Mind and Computers*, pages 115–131, 1984.
13. B. Fluri, J. Zuberbühler, and H. C. Gall. Recommending method invocation context changes. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, RSSE '08, pages 1–5, New York, NY, USA, 2008. ACM.
14. S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time autocompletion of refactorings. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 222–232, Piscataway, NJ, USA, 2012. IEEE Press.
15. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
16. X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 211–221, Piscataway, NJ, USA, 2012. IEEE Press.
17. C. Gentilviso. The 50 worst inventions. Time Magazine, May 27, 2010.
18. L. Hiew. Assisted detection of duplicate bug reports. Master's thesis, The University Of British Columbia, 2006.

19. D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):741 –748, sept.-oct. 2006.
20. E. Horvitz, A. Jacobs, and D. Hovel. Attention-sensitive alerting. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, UAI'99, pages 305–313, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
21. R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne. Controlled experiments on the web: survey and practical guide. *Data Min. Knowl. Discov.*, 18(1):140–181, Feb. 2009.
22. J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. Grouplens: applying collaborative filtering to usenet news. *Commun. ACM*, 40(3):77–87, Mar. 1997.
23. J. A. Konstan and J. Riedl. Recommender systems: from algorithms to user experience. *User Model. User-Adapt. Interact.*, 22(1-2):101–123, 2012.
24. S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification*, pages 712–717. Springer, 2012.
25. T. D. LaToza and B. A. Myers. Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, PLATEAU '11, pages 45–50, New York, NY, USA, 2011. ACM.
26. W. Maalej and A. Sahm. Assisting engineers in switching artifacts by using task semantic and interaction history. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 59–63, New York, NY, USA, 2010. ACM.
27. D. Maulsby, S. Greenberg, and R. Mander. Prototyping an intelligent agent through wizard of oz. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 277–284, New York, NY, USA, 1993. ACM.
28. D. McFarlane and K. Latorella. The scope and importance of human interruption in human-computer interaction design. *Human-Computer Interaction*, 17(1):1–61, 2002.
29. D. McSherry. Explanation in recommender systems. *Artif. Intell. Rev.*, 24(2):179–197, 2005.
30. K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 669–682, Tucson, Arizona, USA, October 2012.
31. E. Murphy-Hill and A. P. Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 5–14, New York, NY, USA, 2010. ACM.
32. E. Murphy-Hill and G. C. Murphy. Peer interaction effectively, yet infrequently, enables programmers to discover new tools. In *Proc. of the 2011 conference on Computer supported cooperative work*, CSCW '11, pages 405–414, 2011.
33. C. Nass and C. Yen. *The man who lied to his laptop: what machines teach us about human relationships*. Current Hardcover, 2010.
34. J. Nielsen. Ten usability heuristics. *Alertbox*, 2005.
35. J. Nielsen. Progressive disclosure. http://www.nngroup.com/articles/progressive-disclosure/, December 2006.
36. J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, pages 249–256. ACM, 1990.
37. N. Niu, F. Yang, J.-R. Cheng, and S. Reddivari. A cost-benefit approach to recommending conflict resolution for parallel software development. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 21 –25, june 2012.
38. D. A. Norman. Affordance, conventions, and design. *interactions*, 6(3):38–43, May 1999.
39. A. A. Ozok, Q. Fan, and A. F. Norcio. Design guidelines for effective recommender system interfaces based on a usability criteria conceptual model: results from a college student population. *Behaviour & Information Technology*, 29(1):57–83, 2010.
40. P. Pu and L. Chen. Trust building with explanation interfaces. In *Proceedings of the 11th international conference on Intelligent user interfaces*, pages 93–100. ACM, 2006.

41. P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.
42. N. Sawadsky and G. C. Murphy. Fishtail: from task context to source code examples. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, TOPI '11, pages 48–51, New York, NY, USA, 2011. ACM.
43. N. Sawadsky, G. C. Murphy, and R. Jiresal. Reverb: recommending code-related web pages. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 812–821, Piscataway, NJ, USA, 2013. IEEE Press.
44. J. B. Schafer, J. Konstan, and J. Riedi. Recommender systems in e-commerce. In *EC '99: Proc. of the 1st ACM Conference on Electronic Commerce*, pages 158–166, New York, NY, USA, 1999. ACM.
45. R. Sinha and K. Swearingen. The role of transparency in recommender systems. In *CHI '02: Extended Abstracts on Human Factors in Computing Systems*, pages 830–831, New York, NY, USA, 2002. ACM.
46. A. Thies and C. Roth. Recommending rename refactorings. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 1–5, New York, NY, USA, 2010. ACM.
47. N. Tintarev and J. Masthoff. Evaluating the effectiveness of explanations for recommender systems - methodological issues and empirical studies on the impact of personalization. *User Model. User-Adapt. Interact.*, 22(4-5):399–439, 2012.
48. M. A. Toleman and J. Welsh. Systematic evaluation of design choices for software development tools. *Software - Concepts and Tools*, 19(3):109–121, 1998.
49. J. Trumper and J. Dollner. Extending recommendation systems with software maps. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 92 –96, june 2012.
50. P. Viriyakattiyaporn and G. C. Murphy. Improving program navigation with an active help system. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '10, pages 27–41, Riverton, NJ, USA, 2010. IBM Corp.
51. C. Wharton, J. Rieman, C. Lewis, and P. Polson. The cognitive walkthrough method: A practitioners guide. *Usability inspection methods*, pages 105–140, 1994.
52. B. Whitworth. Polite computing. *Behaviour & Information Technology*, 24(5):353–363, 2005.
53. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
54. J. Xiao, R. Catrambone, and J. Stasko. Be quiet? evaluating proactive and reactive user interface assistants. In *Proceedings of INTERACT*, volume 3, pages 383–390, 2003.
55. J. Xie, H. Lipford, and B.-T. Chu. Evaluating interactive support for secure programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2707–2716, New York, NY, USA, 2012. ACM.
56. A. Zagalsky, O. Barzilay, and A. Yehudai. Example overflow: Using social media for code recommendation. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 38 –42, june 2012.