Programmer-Friendly Refactoring Tools

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Abstract

Tools that perform semi-automated refactoring are currently under-utilized by programmers. If more programmers adopted refactoring tools, software projects could make enormous productivity gains. However, as more advanced refactoring tools are designed, a great chasm widens between how the tools must be used and how programmers want to use them. The proposed research will bridge this chasm by exposing usability guidelines that will direct the design of the next generation of programmer-friendly refactoring tools, so that refactoring tools fit the way programmers behave, not vice-versa.

1. Introduction

Refactoring, the process of changing the structure of code without changing its behavior, is simply a formalization of what programmers have done for a long time. Typically, research on refactoring focuses on object-oriented programs, but refactoring can also be applied to functional [27] and logic languages [33]. Since the focus of this thesis work is tool user interface issues, the language paradigm is somewhat unimportant. However, for the sake of simplicity, I will use Java as an example throughout this proposal.

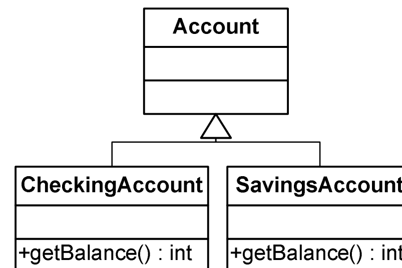
Fowler presents one of the largest catalogs of refactorings to date [20], characterizing 72 different refactorings. Simple ones include Rename Variable, where a variable and all references to that variable are renamed, and Introduce Explaining Variable, where an expression is replaced by a variable, and the expression is assigned to that variable beforehand. For example, the code

```
System.out.println(balance - payment);
```

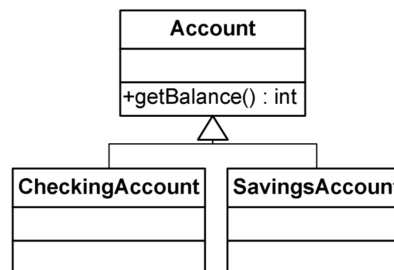
Can be refactored to:

```
int newBalance = balance -  payment;
System.out.println( newBalance );
```

Moderately complex refactorings include Extract Method, where part of the body of a long method is replaced with a call to a new method containing the same code, and Push Up Method, where a method is moved from its subclass to a superclass. For example, these classes



might be refactored to:



Finally, more complex refactorings include Convert Procedural Designs to Objects and Substitute Algorithm, where one algorithm replaces an equivalent algorithm.

1.1. Why Refactoring Is Important

Fowler claims that there are 4 main benefits of refactoring [20], summarized in Sidebar 1. These benefits are based on Fowler's experience.

It is hard to judge how much refactoring is done in practice. Fundamentally, it is impossible to know whether an arbitrary change to a program is behavior preserving. However, Xing and Stroulia report that 70% of structural changes may be due to refactoring [60]. The authors did not detect refactorings that happen at a granularity below the method level (such as Remove Assignments to Parameters or Introduce Explaining Variable), so the actual percentage of refactorings may be much higher.

Other studies have empirically shown refactoring to improve existing code. Kataoka and colleagues have shown that refactoring can measurably decrease a coupling metric in an existing code base [24]. By refactoring using aspects, Benn and colleagues showed that complexity, size, cohesion, and coupling were all improved [11]. Kolb and colleagues showed maintainability and reusability were increased by refactoring existing C code [25]. Geppert and Rossler showed that refactoring can accomplish a specific high-level design goal, as well as unexpectedly improve performance [21]. Ratzinger and colleagues judged that maintainability and evolvability were improved for a large Java program, and that the code produced was much clearer and easier to use [42]. Moser and colleagues showed that software can be made significantly more reusable by refactoring [33]. It appears that refactoring works well in practice.

Refactoring improves the design of software. Agile software methods de-emphasize up-front design [50], so refactoring allows the design to be changed as development goes along. Even with traditional software development methods that start with a good design, changes to requirements and underspecification may introduce a suboptimal design. Refactoring can improve this suboptimal design.

Refactoring makes software easy to understand. If a large piece of software is difficult to understand, maintaining and growing that software is difficult for developers [19]. Refactoring allows programmers to modify an existing program so that they understand it better.

Refactoring helps you find bugs. By restructuring a program, programmers understand more deeply how programs work. Refactoring thus allows the programmer to expose bugs more easily.

Refactoring helps you program faster. The previous points reveal that when programmers spend less time accommodating poor design, understanding code, and tracking down bugs, programmers have more time to develop new functionality.

Sidebar 1. Fowler's four reasons to refactor.

1.2. Refactoring Tools

Refactoring by editing code manually can be a risky endeavor for two reasons:

- **New bugs may be introduced**. When renaming a variable, for example, a programmer may forget to rename some references to that variable, breaking existing code. The problem is more severe when more subtle errors are introduced, such as those that the compiler will not catch.
- **Refactoring may take significant time**. Renaming a method that is called in thousands of places requires searching for all callers and replacing every one with a call to the renamed method.

These risks can be mitigated by using refactoring tools. Most refactoring tools semi-automate the task of refactoring – the programmer chooses some piece of a program to be refactored and the tool performs a specific refactoring. Because the semantics of refactorings is well defined [37], refactoring tools can perform a repetitive and monotonous task on behalf of the programmer. Indeed, Tokuda and Batory estimated that the use of tools decreased restructuring time by 8 and 10 fold, for two existing projects [54]. They also report two additional advantages to using tools: less testing is required because behavior preservation is guaranteed and new designs can be explored quickly.

Because refactoring tools can automatically preserve behavior and be faster than a programmer, programmers should ideally always use refactoring tools, whenever they are available.

1.3. Refactoring Tools are Underutilized

In practice, programmers do not always use refactoring tools when refactoring. However, for the same reason it is difficult to know how much refactoring is practiced in industry, it is difficult to know how often people refactor by hand when they could be refactoring with a tool. However, indirect evidence shows that programmers frequently do not use refactoring tools.

When I performed a controlled experiment involving 16 college senior and graduate object-oriented programming students with a median of 6.5 years of programming experience, only two students said they regularly used refactoring tools in a pre-study survey. Even then, those two students reported they used refactoring tools for only 20% and 60% of their refactorings. Furthermore, of the 37 programmers who have used Eclipse version 3.2 on Portland State University machines before March 2007, only one has used an Eclipse refactoring tool.

Even experienced developers don't always use refactoring tools. I surveyed 112 people at the Agile Open Northwest 2007 conference, where 71 people use environments with refactoring tools. When asked how often they use a refactoring tool when one is available, the mean response was about 68%. The remaining 32% of refactorings are unnecessarily vulnerable to the introduction of errors introduced by refactoring by hand. It seems likely that people actually use refactoring tools less than they report (see Sidebar 2).

When programmers said they use refactoring tools, on average, 68% of the time, the actual percentage of time they use the refactoring tool is probably lower due to bias and a non-representative sample.

Bias. On one hand, of the 71 programmers in my study, 12 said they *always* use refactoring tools. This is highly dubious, considering 6 of them gave reasons why they *do not* use refactoring tools. On the other hand, 4 programmers said they never use refactoring tools. These reports of non-usage can likely be trusted because they only have to remember that they have never used a refactoring tool.

Non-Representative Sample. While the sample size is large, it is probably not representative of all programmers. The conference where the survey was conducted concerned "Agile" software methodology, a kind of software engineering that stresses refactoring as one of its core tenants. Programmers who are not agile may not use refactoring tools as much. Also, the attendees were enthusiastic about software engineering processes, which may inflate their actual usage rates of refactoring tools when compared to non-attendees.

Sidebar 2. Why refactoring tools are probably used less than people report.

Some researchers have observed how often people use refactoring tools, but the data cannot tell us how often people choose not to use refactoring tools. However, by comparing how often people *do* use refactoring tools to how often people would *like* to refactor, we can interpolate whether people are using refactoring tools as much as they would optimally.

Mäntylä and Lassenius [30] asked 37 students to evaluate 10 pieces of code each and decide what refactoring should be applied and how likely they were to refactor that code on a scale from 0 to 5, 0 meaning no refactoring and 5 meaning "refactor immediately." The results show that when people want to perform the Rename refactoring, they give a mean refactoring score of about 3.3 (n=21), but when they want to perform the Extract Method refactoring, they give a mean refactoring score of 4.5 (n=77). If the results are indicative of how programmers behave in the wild, then programmers are much more likely to perform an Extract Method refactoring than a Rename refactoring. This corroborates Fowler's assertion that "Extract Method is one of the most common refactorings I do" [20]. However, Murphy and colleagues' data showed that, of 41 developers over several weeks of observation, there were about 8 times as many uses of the Rename refactoring tool as the Extract Method refactoring tool. In short, Murphy and colleagues show that the Rename tool is much more popular than Extract Method, but Mäntylä and Lassenius show that people would like to do Extract Method much more often than they would like to do Rename. From this we can infer that the Extract Method refactoring tool is not used as much as developers would like[1], and thus developers are forced to either refactor by hand or not refactor at all. It may be the case that developers don't use other refactoring tools as much as they would like either.

Using indirect evidence, it becomes clear that refactoring tools are not utilized as much as they could be, but the extent to which they are underutilized has not been and perhaps cannot be quantified. Despite the theoretical advantages discussed in the last section (speed and accuracy), programmers sometimes don't use refactoring tools.

1.4. Why People Don't Use Refactoring Tools

[1] Another explanation is that the Rename tool is over-used when compared to the Extract Method tool. This is certainly a possibility, but I don't believe that that accounts for the wide disparity between actual Rename tool usage and Extract Method tool usage.

If it has been difficult to quantify how often programmers avoid using refactoring tools, it will be even more difficult to ascertain *why* they don't use the tools.

The most obvious way to determine why a programmer doesn't use tools is to ask them. During Agile Open Northwest 2007, I asked 112 people why they do not use refactoring tools (question and results displayed in Sidebar 3, in the order they were presented on the survey). Let's examine each reason in individually, ordered by popularity.

Answer 1, the tool isn't flexible. This is the most popular response, reported by more than 60% of the respondents who regularly use an environment with refactorings tools. This problem has been addressed in the refactoring research community by creating restructuring scripts [13][44][57], which are discussed in detail in Section 5.3. Essentially, while restructuring scripts may provide the flexibility that programmers desire, existing scripting languages have usability problems, a factor that may decrease, rather than increase, refactoring tool adoption.

Answer 2, lack of knowledge of tool. This also explains the low usage rates of refactoring tools by novices (students) here at Portland State University, discussed in Section 1.3. This problem must ultimately be overcome by educating programmers about the tools, but programmers might be self-educated with sufficiently easy to use tools.

Answer 5, faster by hand. This speed is especially interesting because it runs counter to one of the two main reasons to use refactoring tools, discussed in Section 1.2 (the other being correctness). Creating a fast-enough refactoring tool was one of the three desirable properties of the first refactoring tool, espoused by Roberts and colleagues [45]. However, they only considered the *program transformation time*, not the total, round-trip time to use the tool. Program transformation time is no longer a major concern for users[2]. As I have shown previously [35], how fast a programmer can use a tool depends on a number of factors, but mainly relate to the user interface.

Answer 3, lack of trust. Once again, this problem runs against one of the two reasons to use refactoring tools. But programmers have good reasons to distrust the correctness of refactoring tools. Verbaere has exposed bugs with several of the most popular refactoring tools, bugs where program behavior is modified without notifying the programmer [57]. In December of 2005, I inspected 16 refactoring tools that

When performing a refactoring where a tool is available but you choose not to use it, what usually prevents you? (Please check all that apply)

Answer 1, 44 affirmatives*. The tool isn't flexible enough – it doesn't do quite what I want.*

Answer 2, 26 affirmatives. *I never really learned how to use that particular refactoring tool / I don't know what tool to use.*

Answer 3, 13 affirmatives*. I don't trust the tool to be correct.*

Answer 4, 7 affirmatives*. The tool will probably mutilate my code.*

Answer 5, 24 affirmatives*. I can do it faster by hand.*

Answer 6, 2 affirmatives*. My code base is so large that the refactoring tool takes too long.*

Answer 7*. Other _____________.*

Sidebar 3. Survey responses.

perform the Extract Method refactoring and found that all but two modify program behavior without warning. Given the widespread problem with bugs in most refactoring tools, it is a wonder that more programmers do not distrust tools. More software engineering effort will help solve this problem, but it could also be alleviated by producing simpler toolsets. For instance, by eliminating the standard wizard user-interface for the Extract Method refactoring, the X-Develop [5] and Refactor! [3] tools avoid having to check for several refactoring preconditions. All things being equal, since less code is required to build these tools, they will be less likely to contain bugs.

Answer 4, code mutilation. Seven programmers reported that they would be unhappy with the style of the refactored code. While respondents did not cite specific examples, we can infer that the respondents meant that the transformed code does not adhere to their expectations. For example, the X-Refactory tool [6] may introduce a tuple class when performing an Extract Method refactoring, presuming that the additional class is what the programmer wanted. The programmers' responses agree with Cordy's observation that one of the reasons people don't adopt

[2] If long running refactorings were a major concern, I expect more positive responses on Answer 6.

software maintenance tools (including refactoring tools) is because programmers feel threatened whenever a tool produces some output as if by magic [16]. Essentially, the perception of code mutilation stems from a fear of loss of control. Refactoring tools that give the programmer more control will help assuage this fear.

Answer 6, code base too large. Only two respondents marked this as a problem, which is somewhat surprising, considering the sheer number of legacy, large code bases. The problem here is not that refactoring tools cannot refactor large code bases, it is just that they take an unacceptably long time to do so. Creators of the first refactoring tool discuss how to address this issue practically [45].

In the freeform part of the survey, participants also gave other reasons why they do not use refactoring tools. Two people said "habit." One person said the refactoring menu is too big and searching for the right refactoring takes too long, while another said they avoid GUIs and would only use key bindings to activate refactorings. One person said she preferred "to be aware of the changes myself" and uses the compiler to tell her how to refactor, while another said they find it hard to trust the refactored code, even if it compiles. One person said she usually does multi-step refactorings, but tools can only do one step at a time. Another person said "tools don't do the things I do."

Gauging from people's responses to my survey, it appears that the primary reason why people do not use refactoring tools boils down to usability[3]. If tools could be used consistently with how programmers want to refactor, programmers would use these tools more often. But the survey only indicates some of the usability problems that might exist – pinning down the exact usability problems is the next step.

2. A Programmer's Refactoring Process

In order to define what kind of usability problems exist with refactoring tools, it is necessary to examine how programmers actually refactor – a "programmer's refactoring process." Rather than modifying people's behavior to fit how a tool works, it is better to modify how a tool works to fit how people behave. So let us examine how people behave and later explore how refactoring tools can accommodate this behavior.

[3] The wording of the survey limits the responses to people who don't use refactoring tools only when they are available. Obviously, this precludes a major reason people don't use refactoring tools – they are not available. The research described in this paper concerns how to improve tools that are already available, and the results will extend to tools that have not yet been built.

2.1. When Do People Refactor?

Martin Fowler outlines four different occasions when a programmer should refactor [20]. First, when code is duplicated for the second time, a programmer should factor out the duplication. Second, she should refactor when functionality needs to be added, but the existing code is hard to understand or the addition is not easy to make because of the existing design. Third, she should refactor when a bug needs to be fixed and refactoring the code will help make the code clearer and expose the bug. Finally, she should refactor when programmers are doing a code review and refactoring will immediately produce code that everyone understands. All of these different occasions to refactor are similar in that they occur intermixed with other software engineering activities, in a demand-driven manner. This kind of refactoring, used frequently and consistently to help maintain healthy code, I shall call ***floss refactoring***. An important characteristic of floss refactoring is that a programmer always knows what needs to be refactored, because the programmer is only refactoring that which is hindering her progress.

A related refactoring process occurs when time constraints prohibit immediate restructuring. This refactoring process, occurring after a normal development process completes but the knowledge of what "should have been done" is still fresh, I shall call ***delayed floss refactoring***. The important characteristic of delayed floss refactoring is that it is not mixed with other development practices, but programmers still know what to refactor. Shore discusses this type of refactoring in a forthcoming book [50] with the Agile practice of slack, which permits developers to pay off existing technical debt. Slack is set aside at the end of a development iteration to allow developers to make progress on items not directly achieving their goals, such as by improving software through refactoring.

Another refactoring strategy is to do it in clumps, by setting aside specific time, apart from normal software development. Kataoka and colleagues [24] describe this process, and Pizka [41] reports how an instance of this process worked in practice. This process of refactoring, used after code has become unhealthy and requires correction to become healthy again, I shall call ***root canal refactoring.*** An important characteristic of root canal refactoring is that a programmer does *not* know what needs to be refactored, only that a program

has been hard to maintain in the past and an improved design is desired to accommodate anticipated (but not specified) future changes.

This research will be restricted to floss refactoring, because I believe it to be the most important refactoring process, both in actual practice and in ideal practice, because:

- Root canal refactoring may not be adequate to maintain healthy software. Fowler cautions against it, noting "in almost all cases, I'm opposed to setting aside time to do refactoring" [20]. Pizka describes a 5 month case study where time was set aside to refactor a legacy system. The author reports that root canal refactoring is "harder to apply than expected, …both time consuming and error prone, … [and] the usefulness of refactoring for restructuring purposes without concrete need is doubtful because it is unclear whether the increased beauty will simplify or aggravate future changes." Pizka concludes, "Refactoring is definitely useful in many situations but of minor help for a large scale consolidation effort" [41].
- Floss refactoring is a central tenant of Agile software engineering methods [59]. Agile methods are growing in adoption [7], and likewise I expect floss refactoring to grow.
- Although slack is a feature of Agile development, Shore notes that refactoring new code is too important to be delayed [50]. Delayed floss refactoring only occurs when there isn't enough time to refactor, that is, when a programmer can't refactor fast enough. If tools helped people refactor fast enough (that is, making it faster to refactor than to add, fix, or read code in a bad design), then this kind of refactoring would not be necessary.
- It appears that in practice, there are very few software iterations where the only changes made are refactorings. Such a pure-refactoring iteration would suggest an instance of either delayed floss or root canal refactoring. Weißgerber and Diehl mined the CVS repositories of three large open source software projects, and found no days of pure refactorings [58]. They note that "This is quite surprising, as we would expect that at least in small projects like JUNIT there are phases in a project when only refactorings have been done to enhance the program structure" [58]. Although this evidence is far from conclusive (see Section 5), it does suggest that delayed floss and root canal refactoring are practiced infrequently.
- Likewise, inspecting Murphy and colleagues' data [34], I have ascertained how often refactoring tools are used between version control commits (what I will call an iteration) when compared to the number of source code edits. For 2,672 commits for 31 users, there are 283 iterations that contain at least one refactoring operation. All except 9 of those 283 iterations contained multiple edits to source code[4], suggesting that there are few iterations of pure refactorings[5], and therefore few instances of delayed floss or root canal refactoring.

These points are not meant to portray delayed root canal refactoring as universally unnecessary, but only to justify my choice in focusing on floss refactoring. Having narrowed my research focus, let's examine the floss refactoring process in detail.

2.2. How Does Refactoring Mix with Other Development?

During floss refactoring, with whatever frequency refactoring is performed, refactorings are mixed with other tasks, such as adding functionality or fixing bugs. We can view this as a mix of semantics preserving transformation and semantics changing transformations. Large refactorings, those that are composed out of smaller refactorings, may be spread out over time, so that semantics changing operations are inter-mixed [28]. This allows programmers to perform time-consuming refactorings during the normal flow of development.

2.3. How Do People Refactor?

Using a refactoring tool to perform a refactoring, at the level of a single composite refactoring (one refactoring composed of smaller refactorings), I have developed a model of what happens during successful refactorings (Figure 1). This model is based on my own observations of programmers performing the Extract Method refactoring in Eclipse [35], Fowler's book [20], Lippert's description of "large refactorings" [28], and Kataoka and colleagues' description of a refactoring process [24]. To illustrate this process, let's work through Fowler's Move Method example

[4] Of those 9 edit-less iterations, I suspect most are false positives. In 8 instances, subjects used third-party editors for which edit events are not captured.

[5] The monitoring tool is unable to capture refactorings performed by hand, that is, some edit events may actually represent refactorings. Therefore, subjects may have had refactoring-only iterations that the data cannot show.

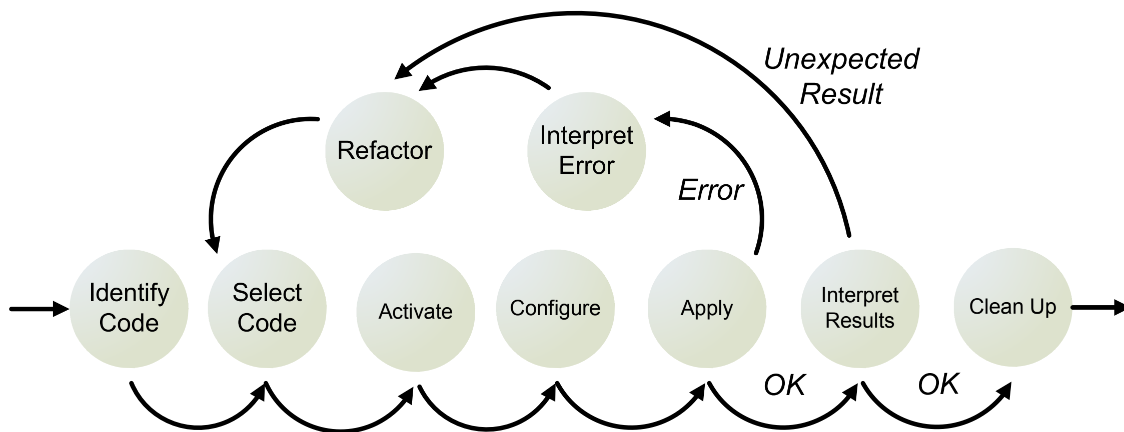


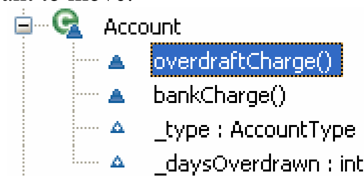
Figure 1. Refactoring with a Tool.

[20] in a modern development environment, Eclipse. Suppose we are working on the following class:

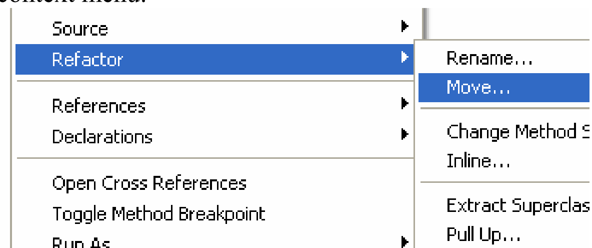
```
class Account {

 double overdraftCharge() {
  if (_type.isPremium()) {
   double result = 10;
   if (_daysOverdrawn > 7)
    result +=
      (_daysOverdrawn - 7) * 0.85;
   return result;
  } else
   return _daysOverdrawn * 1.75;
 }

 double bankCharge() {
  double result = 4.5;
  if (_daysOverdrawn > 0)
    result += overdraftCharge();
  return result;
 }

 AccountType _type;
 int _daysOverdrawn;
}
```

Suppose we are about to add several new account types using the Eclipse environment, all of which have different ways to calculate overdraft charges. Making this change is not easy for the given code, because we would have to add a new case to `overdraftCharge()`. Instead, it makes sense to use let each `AccountType` do the calculation.

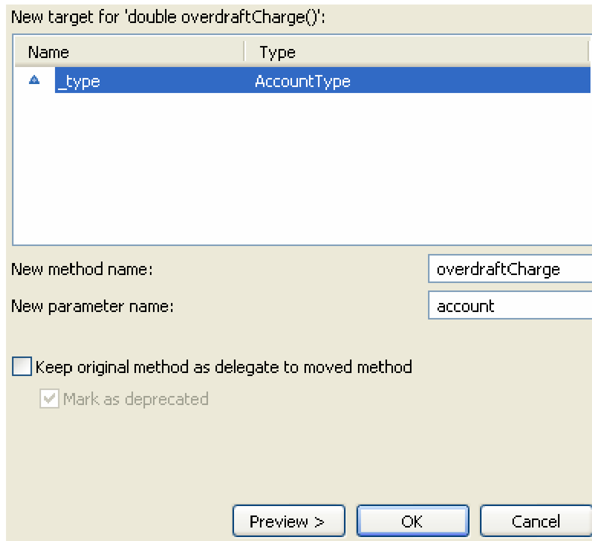
The first step is to **Identify Code** to be refactored. In this case, we want to move the `overdraftCharge()` method into the `AccountType` class. The next step is to **Select the Code** to be refactored. In Eclipse, we can select the code in the outline view by clicking on the method we want to move:



Alternately, we could select the code by simply clicking selecting the text in the editor. Next, we **Activate** the refactoring, usually with a shortcut key or context menu:

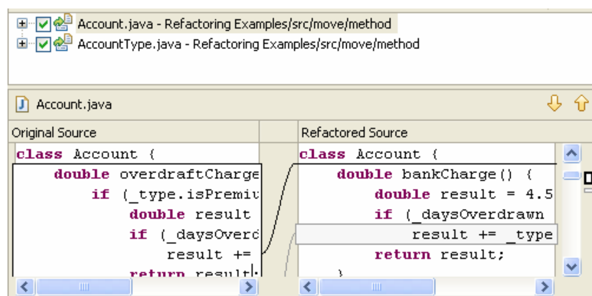


Then, we **Configure** the refactoring. In this case, we have to at least specify what class we are moving the method into. In Eclipse, a wizard is shown, giving us other configuration options as well:



After we are done configuring the refactoring, we **Apply** the refactoring, here by pressing "OK". The refactoring application may either succeed or fail. On failure, the refactoring cannot be applied because a precondition is not met. For example, if `AccountType` already had a method called `overdraftCharge`, the tool would present an error[6]. We would then have to **Interpret the Error**, decide what the error means and how to correct for it. To make the refactoring go forward, we could recurse and **Refactor** again. In this case, we might rename the existing `overdraftCharge()` to a new name, then try the Move Method refactoring again.

However, since we did not violate a precondition in our example, we will usually want to **Interpret Results**, to make sure that the transformation that was applied results in the code we expect. We can either inspect the code directly, or tools may provide a special way to look at the changes, such as a difference view:



[6] Or at least it should. At the time of this writing, Eclipse 3.2 Move Method refactoring tool will break existing code.

When interpreting results, we notice that the resulting transformation was not what Fowler had intended:

```
double overdraftCharge(Account act) {
 if (isPremium()) {
  double result = 10;
  if (act._daysOverdrawn > 7)
   result +=
     (act._daysOverdrawn - 7) * 0.85;
  return result;
 } else
 return account._daysOverdrawn*1.75;
}
```

Instead, Fowler wanted to pass in the `_daysOverdrawn` field from the Account object, like so:

```
double overdraftCharge
             (int daysOverdrawn) {
 if (isPremium()) {
  double result = 10;
  if (daysOverdrawn > 7)
   result +=
     (daysOverdrawn - 7) * 0.85;
  return result;
 } else
 return daysOverdrawn * 1.75;
}
```

To achieve this result, it would be necessary to recurse and **Refactor** again, by performing an Extract Local Variable on `account._daysOverdrawn` in the original code, performing Extract Method on the rest of `overdraftCharge()`, then performing the Move Method again. After this, overdraftCharge() is now correctly moved into `AccountType`, but we are left with the following in `Account`:

```
double overdraftCharge() {
 int daysOverdrawn = _daysOverdrawn;
 return _type.
   overdraftCharge(daysOverdrawn);
}

double bankCharge() {
  double result = 4.5;
  if (_daysOverdrawn > 0)
    result += overdraftCharge();
  return result;
}
```

Now, we need to **Clean Up**, undoing the recursive refactoring steps above (what we might call transient refactorings, since they exist only to allow us to

perform other refactorings), by inlining daysOverdrawn:

```
double overdraftCharge() {
 return _type.
   overdraftCharge(_daysOverdrawn);
}
```

And then inlining overdraftCharge():

```
double bankCharge() {
  double result = 4.5;
  if (_daysOverdrawn > 0)
   result += _type.
    overdraftCharge(_daysOverdrawn);
  return result;
}
```

The code that remains is what Fowler described after refactoring by hand.

In this section, I have outlined how programmers generally use refactoring tools. While exceptions to this process exist (see Sidebar 4), this process represents how a programmer interacts with the original Smalltalk Refactoring Browser [45]. Since its creation over 20 years ago, most tools support this kind of interaction.

Now that we understand when people refactor and how they use tools, we can evaluate how well tools support refactoring, from a human point of view. This will allow us to examine the shortcomings of existing tools and propose some alternative interfaces.

Variations on a Programmer's Refactoring Process

The process described in Figure 1 displays the general use of a refactoring tool. However, in different tools and different environments, the process may change.

A different way to use tools to perform refactorings is to perform the low-level refactorings first, then perform the high level refactorings. This requires greater foresight on the part of the programmer, but can also be more productive. By analogy, if this describes small-step semantics, then Figure 1 describes big step semantics [40].

Also, some steps can be removed or reordered. For example, Configure Refactoring is unnecessary for Extract Local Variable (demonstrated in Section 1). In Extract Method, Configure Refactoring can occur after Apply Refactoring, by first performing the Extraction and then prompting the programmer for a new method name.

Sidebar 4. Survey responses.

3. Problems, Possible Solutions, and Proposed Work

To identify what specific problems exist when using refactoring tools, let's look at each individual step described in Section 2.3, observe shortcomings of existing tools that support that step, suggest some properties of a theoretical, idealized tool, and finally propose thesis work.

3.1. Identify Code

In order to begin refactoring, a programmer has to identify code to be refactored. During floss refactoring, a programmer is adding, fixing, or reading code, so the code to be refactored is simply the code that is hindering progress. If the programmer knows what needs to be refactored to make progress efficiently, then tool support for identifying code to be refactored is unnecessary. In some cases, though, the programmer may not know how the code she is looking at should be refactored

For instance, a programmer may know that the method she is looking at is difficult to understand, but what is making the code difficult to understand is not apparent. As an experiment, Mäntylä and Lassenius asked 36 programmers to say what they thought was wrong with several methods and how they might refactor the methods [30]. Problems such as a long method or long statements were often agreed upon by many programmers, but others such as duplication and "wrong method location" were exposed by only a few programmers. If a tool can help a programmer locate what needs to be refactored, especially code that is difficult to spot with the naked eye, then tools can assist in this part of the refactoring process.

Tools that help the programmer look for such design flaws are called "smell detectors." Fowler originally proposed smells as ways to determine what code should be refactored [20]. Van Emden and Moonen created one of the first tools to automate smell detection, called jCosmo [55]. As discussed in detail in Section 5, jCosmo is unsuitable for floss refactoring because of its long, time consuming tool chain and its

representation of smells at a system-wide level rather than a local code level. Van Emden and Moonen do state that smells encountered during floss refactoring can be displayed “without much intrusion by treating smells similarly to compilation errors or warnings” [55].

Therefore, a student of Moonen named Slinger has described a smell detection tool that presents smells in a compilation error-like form in the Eclipse environment [51]. Essentially, source code is underlined when a code smell is present, as the author is working on it. Bhattacharrya has demonstrated a similar tool [10], while other tools such as IntilliJ IDEA [2] and TogetherJ [4] detect code smells, but present them in the same way which they present code metrics. Discussed in detail in Section 5, I believe these approaches don’t work well for floss refactoring because code smells are not binary like compiler errors and because the programmer may be overwhelmed with false positives. Indeed, Slinger’s use case describes root canal refactoring, not floss refactoring; “When deciding where to begin the refactoring of a software system , the class that contains the most code smells is often seen as the best place to start” [51].

While proposals for code smell detectors for root canal refactoring seem pervasive, there are a few code smell detectors for floss refactoring. Hayashi and colleagues observe that global smell detection tools, such as jCosmo, may distract the programmer from the piece of code she is working on [23]. Instead, Hayashi and colleagues describe a tool that detects two code smells from user edits and suggests refactorings for the code on which the programmer is working. Likewise, Parnin and Görg create a tool for detecting code smells while developers are coding, noting that “developers do not always have a clear time allocated toward solely performing software inspection; instead, inspection is often interleaved with other software development activities” [39]. Both Hayashi and colleagues’ and Parnin and Görg’s tools are discussed in Section 5. In short, while these tools show promise in helping programmers perform floss refactorings, neither is empirically evaluated, nor is empirical validation planned in the future[7].

In order to improve tools that allow programmers to identify code to refactor, the ideal tool has the following user interface properties:

- Report only those smells related to the code the programmer is currently inspecting; For example, if the method the programmer is looking at is too long, the tool will tell her, but not inform her that methods in other classes are too long. However, if a piece of code is duplicated both in the method the programmer is looking at and in a seemingly unrelated class, both pieces of code will be brought to the programmer’s attention.
- Reported smells should not be overwhelming to the programmer, and adding new smells will not increase the interface complexity. For example, the tool might show only the top 3 worst smells in a method, for some definition of “worst.”
- Detection of smells should be done on the fly, similar to incremental compilation, so the programmer does not have to wait or be interrupted to know what smells.
- Smell information should be readily available, but will not be distracting or encourage premature refactoring.
- Present non-trivial smell information. For instance, a tool that points out a Long Parameter List based on a certain threshold is rather worthless – a programmer can clearly see that there are a lot of parameters. Instead, the tool might say that there are several methods with many of the same Long Parameter Lists.

For this thesis research, I will build a tool with these properties. No user study will be conducted to assess how programmers use existing tools because existing tools for smell detection are not widely used in practice, and therefore such a study will be highly contrived. Furthermore, such a study will be of limited value to identify the faults of existing tools, as the tools are still in their infancy and many faults were easily exposed in this section. The tool will be built and refined based on feedback from programmers with the Portland State Computer Science department. I will attempt to implement as many smells as is reasonable in this smell detector; Slinger implements about a dozen [51], so I will attempt to follow suit. I will evaluate the usability of this tool by showing how it can help the programmer expose design defects more quickly and accurately than using existing tools[8].

[7] From personal communication with authors

[8] Although Slinger’s tool [51] is not available, a cheap clone of Slinger’s tool can be built using Eclipse’s TPTP platform; I have already built a prototype smell detector 3 smells in about an hour.

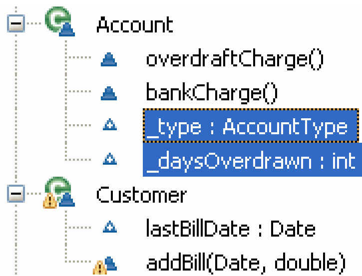
3.2. Select Code

In order for a refactoring tool to perform a restructuring, the programmer must communicate to the environment what should be refactored.

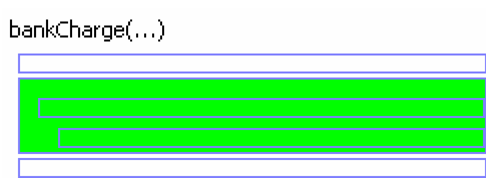
Most refactoring tools allow the programmer to say what should be refactored by selecting code in an editor, with a mouse or the keyboard. Tools may assist in this process, such as hotkeys that select a program statement or abstract syntax tree node for you, and tools that provide a cue to the structure of the program, such as Selection Assist [35]:

```
double result = 4.5;
if (_daysOverdrawn > 0)
    result += overdraftCharge();
return result;
```

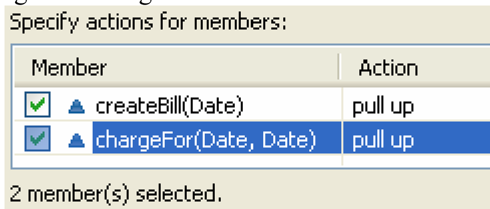
Sometimes, a different view of the program source code allows a refactoring to apply to multiple entities, by selecting more than one. To move several class variables, in Eclipse we can select them in the Outline View:



Likewise, a programmer can select several statements in preparation for Extract Method using Box View [35]:



Selection can also be made later during the Configuration stage:



Code can also be selected by using a program query, such as with iXj [13] or the Refactoring Browser's pattern matcher [44]:

```
``@receiver setBanana: ``@arg
```

This query will match all calls to `setBanana`.

Selecting code for refactoring could be difficult for several reasons. As I have shown in previous work, programmers sometimes have difficulty accurately selecting code, at least for the Extract Method refactoring [35]. The wide variety of ways to select code for refactoring in a single environment may make it confusing for the programmer; for example, in Eclipse, all refactorings can be invoked after selecting code in the editor, but only some can be invoked with the Outline View. Programmers may waste time by losing mental context when switching between views to make selections [17].

In order to improve tools that perform code selection for refactoring, an ideal tool has the following user interface properties:

- It should be impossible to make an invalid selection for the refactoring a programmer wants to perform. At the same time, the tool should not make assumptions about what the programmer meant, such as by changing the user's selection to what the tool considers a valid selection.
- Switching views should be minimal or unnecessary.
- The selection tool should be consistent for all refactorings. The programmer shouldn't have to use one tool for one refactoring and another tool for another.
- When selecting multiple items for a refactoring (such as statements for Extract Method) or refactorings (such as several methods to Pull Up), the time to select all items should be linear for the programmer[9].

For this thesis research, I will build a tool to perform selection. Unlike my previous work [35], this tool will enable the programmer to uniformly select code for several refactorings, as well as selecting several pieces of code to be refactored in one step. I may conduct a study to determine how programmers select and activate several repetitive refactorings in series. The tool will be built and refined with

[9] It might be argued that selection time should be constant, such as can be achieved with program queries. However, I counter-argue that the programmer is going to want to review every primary element (that is, non-referential) that is refactored anyway, which will take linear time.

department and community input. It is critical that this tool enable selection for all or most refactorings, to demonstrate generality. This tool will be evaluated by demonstrating both how single and multiple pieces of code can be selected more quickly and accurately than existing tools. The evaluation will likely have programmers compare this new tool to existing tools (such as Eclipse's Outline View and Java editor) in a controlled experiment.

3.3. Activate

A programmer needs some way to communicate that the restructuring should be performed by the tool. Eclipse has several ways to activate a refactoring: key bindings, context sensitive menus, pull down menus, or implicitly through drag and drop. Murphy and colleagues report that the activation method varies from refactoring to refactoring [34]. For example, the Inline refactoring is almost always activated with a key binding, but the Pull Up refactoring is never activated with a key binding. Why one activation mechanism is preferable to another is unclear.

In my study of several programmers performing Extract Method [35], I observed that programmers have few problems activating a refactoring, although the key combination to activate the refactoring was sometimes difficult for programmers to remember. This might be attributed to less than mnemonic series of key presses. For instance, when a programmer knows what restructuring she wants to perform in Eclipse, she has to map that restructuring (such as dividing up a method) to a refactoring name (such as "Extract Method") to a key binding (such as Alt-Shift-M, for apparently "method," not "move"). This three-level mapping can be difficult for programmers. Another possible problem with activation is that tools are sometimes non-responsive to activation requests. For instance, the refactoring context menu in Eclipse's Java editor changes depending on the selection, so if the refactoring that the programmer wants to perform does not appear, that programmer can come to any number of conclusions: the environment doesn't support the refactoring she wants to do, she hasn't selected the program element that the refactoring tool expects, or perhaps the refactoring tool is simply broken. Worse yet, other environments such as Refactor! [3] hide the refactoring menu whenever a refactoring precondition is violated without explaining what precondition is violated or why.

The ideal tool for activating refactorings should have the following user interface properties:

- All refactorings should be consistently activated in the same way.
- How to activate a refactoring should be mnemonic.
- Upon activation the refactoring tool should provide some feedback, either commencing the refactoring operation or informing the user why the refactoring didn't take place.

For this thesis research, I will build a tool to activate Eclipse's refactorings based on these properties. Rather than being a name-based activation, activation will be structural and gestural. For example, dragging an expression up to an empty line would indicate the user wishes to Extract Temporary Variable. No initial user task analysis will be done to demonstrate the activation problem, as prior work on hotkeys and menus show the disadvantages of both [18]. The tool will be refined based on input from colleagues. It is critical that this tool be implemented for as many refactorings as possible – I will attempt to allow the programmer to activate all refactorings enabled in Eclipse this way. The tool will be evaluated by comparing activation time and memory recall for this tool, hot keys, and menus.

3.4. Configure

An investigation into good interfaces for refactoring configuration is outside of the scope of this thesis proposal. If researched, this stage could yield guidelines for how much configuration the programmer should be required to supply. However, I feel this will yield a less significant contribution than other refactoring stages.

3.5. Apply

No research in this thesis will be done for this stage. An experiment could be used to determine whether or not programmers are concerned when a refactoring tool excessively modifies their code, but obtaining the results is too low a priority.

3.6. Interpret Results

Interpreting the results of a refactoring allows a programmer to make sure that the transformation that the tool applied was what she expected, or at least what she can live with. Primitively, the programmer can do this by inspecting the new source code and comparing it to the old source code (either as remembered, or as archived). Tools may automate this task, such as in

Eclipse, where the tool shows the programmer a difference view of the original and modified source code. Another method for interpreting the results is to repeatedly apply and undo the refactoring.

The process of interpreting the results may be problematic for several reasons. When comparing the results manually to old source code, the programmer is burdened with having to preserve a backup (or remember old code!) before every refactoring. The programmer may lose context when a diff view is presented because it is generally presented as a separate window. However, if the original code window is annotated to reflect the refactoring, the programmer may not be aware of non-local changes made by the refactoring engine.

The ideal tool for helping a programmer to interpret the results of a refactoring should have the following user interface properties:

- Instill in the programmer a sense of trust in the transformation engine.
- Do not disorient the programmer by presenting new views unnecessarily.
- Make the programmer aware that non-local changes were made, as well as the extent of those changes, and give the programmer specific details if she desires.

I propose to build two tools to help programmers interpret results. One tool will give the programmer an at-a-glance, non-modal summary of the extent of a single refactoring. For instance, suppose the programmer uses ExtractMethod. The tool would mark all changed code with a different color:

```
int foo(){
  int value = exMethod(MAX);
  return value;
}

int exMethod(int x) {
  return x^3;
}
```

Another non-modal window will give some information about the change, with hyperlinks for more information:

①	1 code segment modified.
µ	Local changes only.

The second tool will give allow the programmer to view a group of related refactorings at once. This would be accomplished using similar at-a-glance methods as the last tool, and with the difference view. An initial study of how programmers use and understand the standard Unix diff view will be conducted. The two tools will be built with programmer feedback. The tools should work for any number of refactorings. The tools will be evaluated with a user study demonstrating whether programmers understand the scope of changes presented by these tools versus the Eclipse difference tool.

3.7. Interpret Errors

When it is not possible to perform a restructuring without changing program behavior, because a refactoring precondition is violated, the programmer must be told why. This information can be used by the programmer to perform some changes to the code to allow the refactoring to proceed. To my knowledge, every refactoring tool, with the exception of Refactoring Annotations [35], represents the violated precondition information as a textual error message, similar to a compiler error message.

As I have described previously [35], there are several problems with the current representations of these errors, all relating to programmer misunderstanding error messages. One problem is that programmers rarely read these error messages, and when they do, they can interpret them incorrectly. Programmers sometimes believe an error message to be the same as a different error message, simply because the textual error messages look similar. Programmers sometimes cannot understand the meaning of the error messages, probably because the messages are vague and do not refer to concrete source code. The most disturbing problem, in my view, is that the error messages can be directly linked to tool non-adoption. For example, a programmer sees an error message (for example, "illegal break statement"), interprets it broadly (for example, "I can never refactor when a break statement is present"), and then later avoids trying to refactor code that violates the programmers interpretation, even if later refactorings are perfectly acceptable. If a refactoring tool allows a programmer to easily misinterpret an error message, the programmer may not use that tool to its full potential in the future.

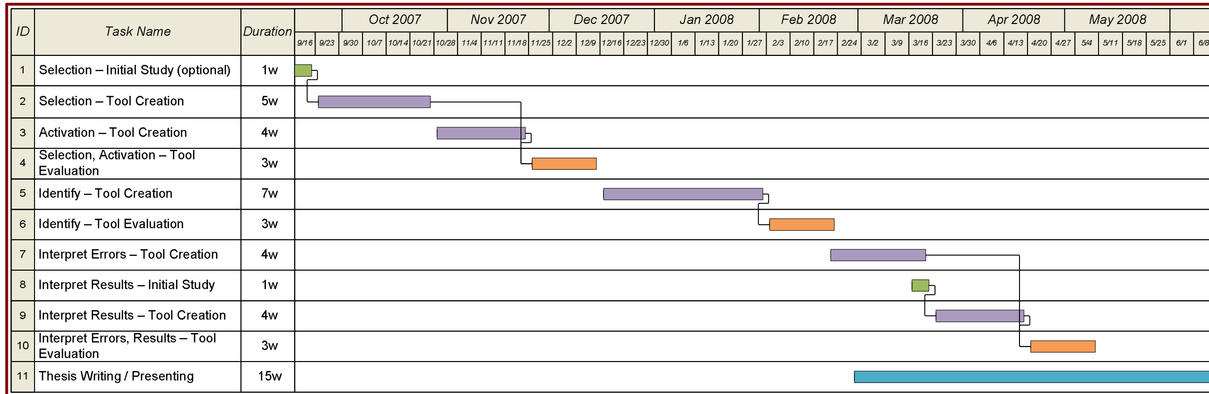


Figure 2. Thesis work timeline.

Based on my preliminary work [35], the ideal refactoring tool should present violations of refactoring preconditions with the following user interface qualities:

- Be specific about the location(s) of violations.
- Show every violated refactoring precondition.
- Allow the programmer to easily distinguish violated preconditions (show stoppers) from warnings and advisories. The programmer should be able to tell if something went wrong at a glance.
- Give an indication of the amount of work required to fix the violated preconditions.
- Display the violation relationally, when appropriate. For instance, when an error is caused by an interaction between a loop and a break statement, indicate both as related to the problem.
- Use different, distinguishable representations for different types of violations.

I propose extending my past research to include explaining violations of other refactoring preconditions. No initial user study will be necessary. The tools will be built by identifying preconditions reported by Opdyke [37] and displaying them in a way consistent with my existing guidelines, above. While this research intends to expand the guidelines to all refactorings, in practice I will only implement precondition explanations for a few refactorings, then show how those explanations are sufficient for the remaining refactorings[10]. The tools that perform the explanation will be evaluated by showing an increase in speed, accuracy, and user satisfaction over traditional text-based explanations.

3.8. (Recursively) Refactor and Clean Up

Discovering how to assist programmers can better organize high-level refactorings and recursive refactorings is outside of the scope of this thesis proposal. If conducted, this research would largely be about a new tool on refactoring task organization. Because a meaningful comparison of such a new tool and existing refactoring tools (with no organizational support) would be difficult, no such research within the scope of this thesis is planned.

4. Research Plan

In the previous section, I proposed several tools that might improve the process of refactoring at each stage in the refactoring process. Each tool will be written in Eclipse, both because I already have familiarity building tools for that environment, and because it facilitates easy comparison to existing tools. As I have already noted in the last section for each tool, some preliminary empirical results may be required to assist in building the tool. Each tool will be evaluated as described in the last section.

Figure 2 gives a timeline estimation of the proposed work. Each work component is represented as a row, where several work components are grouped together when they address the same refactoring stage. The first stage (green) is optional and represents an initial user study, the second stage (purple) represents tool building, and the last stage (orange) represents the comparison of the newly built tool(s) to an existing tool. Some evaluation stages are combined – for example, Selection and Activation tool evaluation will

[10] For example, I have not yet demonstrated an effective explanation for Extract Method naming conflicts, but such an explanation should extend to naming conflicts in other refactorings, such as Move Method.

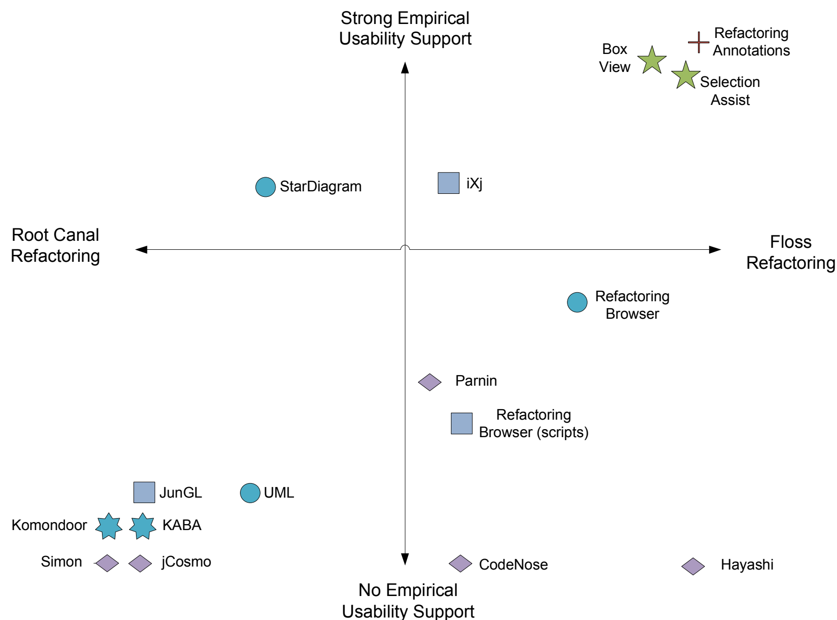


Figure 3. Refactoring tool map.

occur in a single stage. This will allow me to amortize the cost of performing experiments by having human subjects test two different kinds of tools in the same session. I believe that the proposed timeline is reasonable, based on previous experience with the Extract Method refactoring tools.

The qualities of each tool built, comparisons to existing tools, and observations of programmers using these tools will facilitate the ultimate goal of this thesis: to promulgate a set of user-interface guidelines for future refactoring tools.

5. Related Work

In recent years, there has been much ado about refactoring, but little discussion on what tools that support refactoring should look like. In this section, I discuss previous research related to refactoring tools and how people perform refactorings. Most related work suggests a tool to improve the refactoring process.

As a summary, each tool described in this section is mapped into Figure 3 on two axes. On the y-axis, each tool is plotted based on how much empirical evidence for improved usability is provided by the research. On the x-axis, each tool is plotted based on how well it supports root canal refactoring or floss refactoring. Similar tools are denoted with the same marker (for example, iXj and JunGL are both kinds of refactoring scripts, so they are denoted with squares). Placement on this map is subjective.

It is important to notice that most tools provide little or no evidence for improved usability and most are geared towards root canal refactoring. This research will provide evidence for improved usability for tools built to support floss refactoring.

Let's examine related work in detail.

5.1. Code Transformation

5.1.1. Refactoring Preconditions

In his Ph.D. thesis, Opdyke defined a set of common refactorings and the preconditions that must be true in order for the restructuring to preserve behavior [37]. While these preconditions are not necessarily complete for all languages [54], they provide a good starting point for understanding what sorts of errors refactoring tools might want to communicate with the programmer. This thesis laid the formal groundwork necessary for the implementation of tools.

5.1.2. First Tool: The Refactoring Browser

Roberts and colleagues built the first refactoring tool, implemented in Smalltalk, called the Refactoring

Browser [45]. This tool implemented several refactorings discussed by Opdyke [37], including checking preconditions and transforming code. In Roberts' thesis, three desirable properties for refactoring tools are given: speed, undo support, and environmental integration [43].

Roberts' properties seem to be obeyed in contemporary refactoring tools, yet as we have seen, refactoring tools are still likely underused. I attribute this partially to Roberts' limited definition of speed, which included only the time to transform the underlying program, not the time for a programmer to use the tool, start to finish. Of course, being the first refactoring tool, Roberts and colleagues probably had little idea of how other people might use the tools.

5.1.3. Current Tools

Since the first refactoring tool, many development environments have included refactoring tools for many languages. It would be difficult to track down every refactoring tool and characterize the user interface of each[11]. However, in November of 2005, I attempted to review a version of every refactoring tool that performs Extract Method. In total, I evaluated 16 tools:

Product	Version
Refactoring Browser	In VisualWorks 3.7.1
X-Refactory	2.0.8
Ref++	1.2.5
Visual Studio	2005
Idea	5.0.1
Eclipse	3.1.1
Refactor-It	2.5.2
Netbeans	5
J Builder	2005
Code Guide	7
C# Refactory	2.0.4
Resharper	1.5
SlickEdit	10.0.2
Model Maker	8.1
X-Develop	1.1
Refactor! Pro	1.0.26

The review showed that nearly every Extract Method tool has an interface similar to the original Refactoring Browser. That interface is essentially as described in Section 2.3.

A few exceptions to the usual dialog or wizard interface emerged, however. The most significant difference is in Refactor! and X-Develop, where dialog boxes are avoided whenever possible. In these environments, the name of the extracted method is simply updated in the text editor, and references to the extracted method are updated simultaneously. Likewise, the rename refactoring is performed this way in Refactor!, X-Develop, and Eclipse, eliminating the need for a dialog box. Apart from these exceptions, there are no major user interface differences in commercial refactoring tools from the original Refactoring Browser.

5.1.4. Alternative Interface Tools: StarDiagram and UML

While commercial refactoring tools have largely homogenous user interfaces, previous research has suggested two interesting user interfaces for performing program transformation.

About the same time Opdyke was formalizing refactoring, Griswold's thesis discussed behavior-preserving-transformations to Lisp programs [22], although he did not call it "refactoring." Later, Griswold's student Bowdidge implemented the visualization Star Diagram which allowed programmers to visually encapsulate variables [14]. Star Diagram is essentially a graph representation of several abstract syntax trees. Bowdidge described a study of programmers which showed that Star Diagram helped programmers in a small restructuring task. Recently, O'Connor, Shonle, and Griswold showed how Star Diagram can be adapted for Extract Method [36], but no human subjects experiment was conducted.

Star Diagram solves two problems with mainstream refactoring tools. First, selecting valid code to be refactored is fairly straightforward, because nodes on an abstract syntax tree are the only selectable program elements. Second, refactoring several pieces of code at once is sometimes easier, because several abstract syntax trees can be overlaid.

While Griswold and colleagues demonstrated a unique refactoring tool which was shown effective for a small refactoring task, it remains to be seen how many refactorings Star Diagram can support. Furthermore, restructuring using Star Diagram requires that the programmer move away, at least temporarily, from the programmer's usual program view – the source code. Having to switch between an editor and Star Diagram may cause a programmer to lose mental context.

[11] Martin Fowler attempts to keep a list of refactoring tools at www.refactoring.com/tools.html. The list contains 27 refactoring suites, each of which may contain dozens of refactoring tools.

In early work on refactoring, Tokuda and Batory suggested that Unified Modeling Language (UML) was an ideal tool interface for refactoring [54]. Several researchers realized this ideal [9][12][56], although none of them provided evidence that the user experience was improved.

Refactoring UML diagrams may solve problems in the areas of refactoring identification, code selection, activation, and some result interpretation. However, while any refactoring tool can be activated on the source code of a program, only a few refactorings can be activated on UML diagrams. For example, Move Constant is a good candidate for implementation in a UML class-diagram interface, but Extract Method doesn't make any sense on a class-diagram. As Roberts notes, "most refactorings have to manipulate portions of the system that are below the method level" [43].

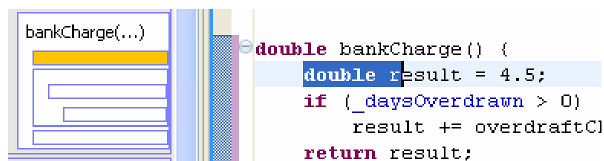
5.2. Selection Assist, Box View, and Refactoring Annotations

Previously, I have researched what makes a good refactoring tool in terms of how code is selected, how configuration information is conveyed, and how violations of refactoring preconditions are displayed [35]. This research was based on observations of programmers performing the Extract Method refactoring.

I built two alternative tools for selecting code to provide proper input to the Extract Method refactoring: Selection Assist and Box View. Using Selection Assist, a programmer simply selects program statements as normal, with a highlight providing a visual cue as to the extent of program statements:

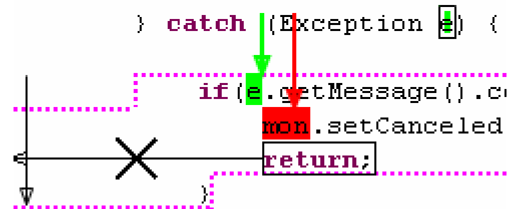
```
double result = 4.5;
if (_daysOverdrawn > 0)
    result += overdraftCharge();
return result;
```

Using Box View, the programmer uses a view adjacent to program code to select statements:



In controlled experiments, I showed that by using Selection Assist or Box View, statement selection speed and accuracy can be improved over the standard keyboard and mouse. Results showed that while no selection tool is strictly superior, each tool was useful in a specific situation. For example, Selection Assist is more helpful during floss refactoring because the programmer doesn't need to change her work habits to use the tool.

I also built a tool that helps with program configuration and preconditions by eliminating the need for some configuration information (which variables are passed into the extracted method and which is returned) and replacing violated precondition error messages with a graphical representation. This was done by creating Refactoring Annotations. For example, suppose the programmer wanted to extract the following code segment and activated Refactoring Annotations:



The refactoring annotations tell the programmer that two variables must be passed in to the Extract Method (`e` and `mon`), but the code cannot be extracted because the piece code returns sometimes, but flows top to bottom others. Experiments showed that programmers understood the causes of violated preconditions several times faster and more accurately using Refactoring Annotations versus standard Eclipse error messages.

Selection Assist, Box View, and Refactoring Annotations all measurably improved the usability of refactoring tools. However, the total impact is limited in that (1) the tools only applied to one refactoring, Extract Method, and (2) the tools only assisted in 3 stages of the refactoring process described in Section 2.3. The research proposed in this thesis is essentially a generalization of my previous work.

5.3. Restructuring Scripts

Rather than invoking refactorings with graphical user interfaces, restructuring scripts allow the programmer to execute a primitive or composite refactoring using a domain specific language. Several authors have devised restructuring scripts for different purposes.

5.3.1. Restructuring Using Tree Pattern Matching

Roberts and Brant recently described how the underlying framework for the original Refactoring Browser can be repurposed to support restructuring using scripts [44]. Although not necessarily behavior preserving, Roberts and Brant's scripts provide a way for the developer to plan and execute a series of restructurings. For example, the following script renames all printString messages and adds a default parameter:

```
`@receiver printString →
      `@reciever printOn: Transcript
```

To use these scripts as refactorings, it is up to the user to verify that all preconditions hold. So while scripts add flexibility to refactoring tools (as requested by users in Section 1.4), scripts require extra vigilance to maintain safety. Furthermore, while these scripts have conceivable benefits over traditional refactoring tools, the authors provide no usability comparison. Rather, they simply state that their pattern language is "difficult to understand and use" and state that the issue remains at the forefront of their research [44].

These scripts are designed to be used over abstract syntax trees, and it appears that higher level restructurings (such as Move Method or even Extract Method) are not possible to express. Because of this, these scripts can perform only a subset of all refactorings.

5.3.2. iXj

Similar to Roberts and Brant's scripting language, Boshernitsan and colleagues have created a scripting language for program restructuring, with an emphasis on an improved user experience [13]. Like Roberts and Brant's work, the user is again in charge of semantics preservation and restricted in the kinds of refactorings that are possible.

In contrast to Roberts and Brant's scripts, the authors claim that their environment, iXj, is "intuitive, easy to learn, and effective" [13]. In a controlled experiment, they asked 5 human subjects to perform 6 pre-defined tasks using their tool. All subjects successfully performed the tasks and reported that they thought the tool was intuitive. The experiment was limited in a number of respects – the most serious of which, for the purposes of this thesis proposal, was that their tool was not compared to any existing tool, refactoring or otherwise.

5.3.3. JunGL

Verbaere and colleagues have proposed a domain specific language called JunGL to express a wide variety of refactorings [57]. As a result, JunGL is considerably more complex than other restructuring scripting languages. The main purpose of JunGL is to build more reliable refactoring tools, not more usable ones. Concordantly, the authors do not provide any evidence that JunGL is easier to use than existing refactoring tools.

5.4. More Automation in Refactoring

Some authors have suggested that more automation in refactoring will improve the tools. More automation, theoretically, means fewer steps that the programmer must perform, and therefore will lead to faster refactoring.

5.4.1. Komondoor and Horotwitz

Komondoor and Horowitz have proposed an algorithm that performs the Extract Method refactoring [26]. Unlike mainstream tools, their algorithm allows the programmer to extract "difficult" statements, statements that include non-local jumps, such as break and return statements. This is achieved by introducing new global variables which are set in the extracted method and checked in the caller. This work represents one way of dealing with violated refactoring preconditions – by manipulating the underlying code automatically so that a refactoring may go forward.

This approach to dealing with refactoring preconditions does not significantly improve user interface issues for three reasons.

First, it introduces extra code that the programmer may not have expected. As I have shown in the refactoring survey (Section 1.4) and as Cordy has noted [16], programmers are sometimes uncomfortable with unanticipated changes to their code.

Second, the automated change is sometimes only one of several possible corrections to the source code. For example, I was recently working on this piece of code:

```
while(true){
 if(!records.next()){
  break;
 }
 Record r2 = new Record(records);
 handleNextEvent(r2);
}
```

Using Komondoor and Horowitz's approach, if I wanted to extract the body of this while loop, the results would look like this:

```
while(true){
  body(records);
  if(DID_BREAK) break;
}
```

However, introducing a little bit of programmer expertise will produce a much cleaner version:

```
while(!records.next()){
  body(records);
}
```

Essentially then, Komondoor and Horowitz's algorithm sometimes replaces good engineering with automation.

Finally, this paper represents one way to deal with one out of six preconditions in one out of dozens of possible refactorings. It represents a clever solution to a relatively small problem – likewise other clever solutions exist for other refactorings, but one must wonder whether they are worth the effort.

5.4.2. Guru and KABA

Moore suggested a tool called Guru to improve Self programs by automatically restructuring inheritence hierarchies [32]. Likewise, Strenchenbach and Snelting proposed a tool for Java called KABA for automatically refactoring class hierarchies [53]. In both systems, given a complete set of clients to a class hierarchy, the refactored hierarchy is claimed to improve locality and cohesion. In KABA, the user is presented with a preview of the resulting hierarchy in a UML class diagram-like view, upon which the user may refactor to her liking.

Both Guru and KABA are applicable only to root-canal refactoring because the tools don't restructure what the programmer sees as an immediate problem; they restructures the whole system, regardless of what the programmer is or was having trouble with. Furthermore, KABA doesn't restructure source code, it restructures Java byte code, so if the programmer wanted to restructure her program, she would have to decompile it. The manual refactoring interface presented by KABA may be an improvement to existing refactoring tools, but no studies involving human subjects were conducted.

5.5. Usability Guidelines

Given that a major contribution of this thesis is a set of guidelines to built better future refactoring tool, it's worth examining why existing guidelines are insufficient.

First, I have found that general usability guidelines can be applied retrospectively to explain why a refactoring tool is good, but they are sometimes impractical for improving the user interface of a refactoring tool. For instance, if we tried to apply Smith and Mosier's five objectives for data display [Smith84] to refactoring precondition violations, we would find that the traditional error messages presented by refactoring tools meet the objectives quite well. However, as my previous research has shown, traditional error messages have many usability problems [35].

Guidelines can be conflicting as well. For instance, one of Smith and Mosier's objectives is to strive for consistency of data display [52], but an axiom provided by Cooper and Reimann is that differently behaving elements should be displayed differently [15]. In my research, it was not obvious how these conflicting requirements should be resolved in a refactoring tool until a prototype was created, based on observed tasks of human subjects [35].

The problem of ambiguous and conflicting general usability guidelines is not unique to refactoring tools. As Schneiderman states, "These underlying principles must be interpreted, refined, and extended for each environment" [46]. In this spirit, the refactoring tool guidelines I propose in this thesis will interpret, refine, and extend general guidelines espoused in previous research.

Second, I have found that specific usability guidelines that might be applied to refactoring tools are currently inadequate. If we investigate usability of refactoring tools at a low level, we can infer some guidelines from analogous tools. For example, investigating the "Interpreting Error" stage from Figure 1, we might learn something from research on error messages for computer systems in general. Schneiderman gives several properties of a good error reporting tool – one of which is user-centered phrasing [47]. The problem with Schneiderman's guidelines is that they are *too* specific – they imply a default representation for errors. In this case, Schneiderman implies that a good representation is textual, although as I have shown previously, a graphical representation of errors is superior in some cases [35].

To my knowledge, there are few existing usability guidelines for refactoring tools. Guidelines might be

gleaned from bug reports [8] and product improvement studies [38], but such guidelines are product specific and seem to suggest only modest improvements to existing tools. As noted before, Roberts and colleagues suggested 3 usability guidelines for refactoring tools [45], but in the ten years that have elapsed, we now have a more complete picture of the tasks in which refactoring tools are used. That picture will help us refine these guidelines.

Although existing refactoring usability guidelines are limited, as this research progresses, I expect that I will find some guidelines from analogous tools helpful. However, to attempt to identify all of these previous guidelines up front is a Herculean task.

Ultimately, if good guidelines were generally known and accepted for refactoring tools, I would expect to see these guidelines manifested in new refactoring tools and consequently see a significant increase in refactoring tool usage. However, since tool adoption is still low, I believe we are still in search of refactoring tool usability guidelines. This thesis work will provide that.

5.5.1. Good Tools

This thesis proposal attempts to answer the question, what makes a good refactoring tool? Likewise, Mealy and Strooper have attempted to answer the same question by evaluating tools functionally [31]. However, evaluation of usability was performed at a very coarse grained level (for instance, whether the interface was a GUI or command-line). Indeed, they conclude that the "usability of refactoring tools requires further research/consideration" [31].

5.6. Refactoring Process

This subsection identifies work that has attempted to catalog the refactoring process, as I have done in Figure 1. This section also characterizes observations and data that other researchers have collected about the refactoring process, as well as the limitations of those studies.

5.6.1. Fowler

Fowler described how refactoring should almost always take place alongside normal development, essentially advocating floss refactoring over root canal refactoring [20]. Fowler's process of refactoring is described in detail in Section 2.1.

5.6.2. Kotaoka and Colleagues

Kataoka and colleagues suggest a refactoring process that is somewhat similar to mine, with the following stages: original source code, bad-smell detection, bad-smell analysis, program weak points, refactoring planning, improvement plan, refactoring deployment, refactoring, refactoring application, and finally improved source code [24]. The process does not appear to be derived empirically, unlike my process, and is straight-line, rather than recursive, so it is probably not exploratory in nature. Accordingly, the authors describe the process as requiring extensive planning and the involvement of three people – a developer, an analyst, and a manager. It appears that this process describes root canal refactoring.

5.6.3. Large Refactorings: Lippert

Lippert describes how large, composite refactorings occur in practice [28]. Lippert describes the process at a high level, where a large refactoring must be executed over several days alongside normal development. This process seems to describe floss refactoring, albeit without a tool.

5.6.4. Slack

Slack is the Agile practice of allowing extra time in the development process for paying off technical debt using various techniques, including refactoring [50]. At first glance, this seems to advocate root canal refactoring. However, since programmers are simply refactoring slightly later than they would have in floss refactoring (they already know what they want to refactor), I would categorize this as delayed floss refactoring. Moreover, refactoring during slack means you failed to refactor during normal development, suggesting that refactoring during slack is the degenerate case (caused by not-fast-enough tools), not the preferred one.

5.6.5. Weißgerber and Diehl

Weißgerber and Diehl analyzed the version history of three open-source projects and found that there were various times when a lot of refactoring activity occurred, but there were no days when only refactorings occurred [58]. This implies that floss refactoring is much more prevalent than delayed floss or root canal refactoring.

However, there are two major limitations to this study. First, the three projects may not be

representative of all software projects. Second, the authors were able to detect only a limited number of refactorings, so some changes that they detected as edits may have in fact have been refactorings. For example, no refactorings below the method level, such as Inline Temporary, were detected.

5.6.6. Murphy and Colleagues

Murphy and colleagues recently discussed observations of 41 programmers using Eclipse [34]. The subjects were studied using an Eclipse plugin, so that every event that occurred (such as a refactoring being invoked) was recorded. The authors reported what percentage of users used each refactoring, and how the refactoring was activated. The authors have made the data available to me, and the data supports the opinion that floss refactoring is the most popular way to refactor.

However, Murphy and colleagues' data is limited in two ways. First, the participants in the study should be considered early adopters as they elected to try the authors' software, and therefore do not represent average programmers. Second, the data reflects only those refactorings invoked using tools. Refactorings that were performed manually could not be recorded.

5.6.7. Mäntylä and Colleagues

In two papers, Mäntylä and Lippert discuss how programmers decide what smells exist in source code and when they refactor, based on human subjects analysis [29][30]. In one paper, Mäntylä and Lippert show the developers agree with each other and with metrics on whether simple smells are present, such as Long Method, but usually disagree on more complex smells, such as Feature Envy [29]. Furthermore, the authors show the programmers often disagree about whether a piece of code should be refactored at all, and the decision cannot be linked to programming experience. In another study, Mäntylä discusses what drives programmers to refactor [30]. The results show that there are many reasons that programmers refactor and that complex issues are exposed by only more experienced developers. The authors also discuss how some code smells, such as Poor Algorithm, are frequently identified as problematic, but are impossible for any tool to detect.

5.7. Code Smell Detection Tools

Code smells were originally proposed by Fowler [20] to determine what code to refactor. In terms of detection, they are like program metrics, but in intended usage, they are much different. Because smells are supposed to be used as a precursor to refactoring, they should be integrated into the refactoring process. As I have demonstrated, since floss refactoring is the more common kind of refactoring, any tool that supports the automatic detection of smells should be optimized for floss refactoring. Such tools, which we will call smell detectors for convenience, are discussed in this subsection.

5.7.1. Simon and Steinbrückner

Simon and Steinbrückner have presented a visualization which depicts cohesion and coupling metrics in order to suggest refactoring [48]. While the tool could be used in either floss or root canal refactoring, it appears the tool is better suited to root canal refactoring, because of the time that must be invested each time the tool is invoked. Essentially, the user must select the classes she feels might be smelly, activate the tool (which may take 15 minutes or more for large projects), interpret the 3 dimensional results in a web browser, then return to the code in order to perform the refactoring, if indeed the code had poor cohesion or high coupling. The authors did not provide human-subject evaluation.

5.7.2. Van Emden and Moonen

Van Emden and Moonen described a smell detector and how authors would use it [55]. The authors distinguish two different uses of a smell detector, one similar to floss refactoring, the other using code smells to help code inspectors or reviewers assess the quality of code. The authors decide to pursue the code assessment use case, so their tool, jCosmo, presents code smells from a high level as a colored tree visualization representing many pieces of source code. This representation, however, is unsuitable for floss refactoring, because during that process, programmers already know what needs to be refactored. jCosmo, then, seems optimized for root canal refactoring. The authors did not provide human-subject evaluation.

5.7.3. Slinger

In follow-up work to jCosmo, Moonen's student Slinger built the other tool – a smell detector called CodeNose to be used by programmers, ostensibly in a floss refactoring context [51]. CodeNose, written as a plugin for Eclipse, presents code smells like compiler

errors. That is, smells are overlaid on source code by underlining the offending code.

Slinger provides no human subjects evaluation, so it is difficult to know for sure whether this user interface is effective. However, several user interface problems might arise in practice. First, compiler errors either exist or they do not exist, but code smells occur in a wide range of values. For instance, a long method may be 10 lines or it may be 1000 lines – intuitively, the 1000 line method is smellier. Secondly, some smells are inherently relational, so simply pointing at a single point in source code as the problem may be futile. For example, Refused Bequest is a smell that concerns both a class and its superclass. Underlining one of them assumes that just one is the problem, and underlining both of them doesn't imply a relationship between the two. Thirdly, simple text overlays may not scale, because every piece of code may have some smell, and some pieces of code may exhibit multiple smells. For instance, comments and message chains are both smells. Underlining every comment and message chain may be overwhelming to the user. Although it is difficult to say with certainty, especially without human subject testing, it appears that this user interface to code smells is impractical.

5.7.4. Parnin and Görg

Parnin and Görg discuss a task-oriented visualization of code smells based on developer interviews [39]. The interviews suggest that existing smell detectors, such as jCosmo, do not fit well with how programmers actually refactor. Instead, the authors introduce a lightweight visualization that can be activated quickly and easily during floss refactoring. Because developers reported that code critic systems (including smell detectors) generated too many results, the authors' tool only generates results for the current task context. However, a programmer must still switch windows which may cause a programmer to lose mental context.

From the short published abstract, it appears that the authors did not conduct human subject evaluation. The authors report that they have no future plans to continue research with the tool.

5.7.5. Hayashi and Colleagues

Hayashi and colleagues also argue that system-wide smell detectors, such as jCosmo, may not present an optimal interface to the programmer; "sudden instruction of refactoring to currently irrelevant parts [of the program] may cause confusion" [23]. In response, the authors describe a tool to suggest refactorings based on program edits. For example, if the programmer cuts and pastes a piece of code to the same class, the tool suggests extracting a method. Thus far, the authors have not provided human subjects evaluation, but the approach appears promising.

6. Expected Contributions

This research is expected to make three contributions:

1. A set of observations about the use of refactoring tools during programming tasks, specifically including deficiencies with current tools.
2. A refactoring toolset built on top of an existing environment, validated by controlled experiments comparing the new tools to existing tools.
3. A set of guidelines for building future refactoring tools.

7. Acknowledgements

This research was supported by the National Science Foundation, grant number CCF-0520346.

8. References

[1] Eclipse. The Eclipse Foundation. 2007.

[2] IDEA. IntelliJ. 2007.

[3] Refactor!. Developer Express. 2007.

[4] TogetherJ. Borland. 2007.

[5] X-Develop. Omnicore Software. 2007.

[6] X-Refactory. Xref-Tech. 2007.

[7] S. Ambler. Agile Adoption Survey: March 2006. http://www.ambysoft.com/surveys/agileMarch2006.html. Accessed March 2007.

[8] T. R. Andersen, "Extract Method: Error Message Should Indicate Offending Variables." https://bugs.eclipse.org/bugs/show_bug.cgi?id=89942, accessed November 2005.

[9] D. Astels, "Refactoring with UML." In Proc. 3rd Int'l Conference on eXtreme Programming and Flexible Processes in Software Engineering. pp. 67-70, Alghero, Italy (2002)

[10] A. Bhattacharrya, and R. Fuhrer. 2004. Smell detection for eclipse. In *Companion To the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA, October 24 - 28, 2004). OOPSLA '04. ACM Press, New York, NY, 22-22.

[11] J. Benn, C. Constantinides, H.K. Padda, K.H. Pedersen, F. Rioux, , and X. Ye. Reasoning on Software Quality Improvement with Aspect-Oriented Refactoring: A Case Study. In W.T. Tsai and M.H. Hamza, editors, Proceedings of the 2005 Software Engineering and Applications, Phoenix, AZ, USA, November 2005.

[12] M. Boger, T. Sturm, and P. Fragemann, "Refactoring Browser for UML," presented at NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, 2003.

[13] M. Boshernitsan, S. Graham, and M. Hearst, "Aligning Development Tools with the Way Programmers Think About Code Changes," presented at 2007 SIGCHI Conference on Human Factors in Computing Systems, 2007.

[14] R. W. Bowdidge, ``Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization," Ph.D. Thesis, Technical Report CS95-457, Department of Computer Science and Engineering, University of California, San Diego, November 1995.

[15] A. Cooper and R. Reimann. About Face 2.0: The Essentials of User Interface Design. John Wiley & Sons, Inc., 2003.

[16] J. R. Cordy, "Comprehending reality - practical barriers to industrial adoption of software maintenance automation," presented at Program Comprehension, 2003. 11th IEEE International Workshop on, 2003.

[17] B. de Alwis, and G. Murphy. Using Visual Momentum to Explain Disorientation in the Eclipse IDE. In Proc. IEEE Symp. Visual Languages and Human Centric Computing 2006.

[18] A. Dix. "Accelerators and toolbars: learning from the menu," Adjunct Proceedings of HCI International, Tokyo, July 1995.

[19] R. Deline, A. Khella, M. Czerwinski, and G. Robertson, "Towards understanding programs through wear-based filtering," presented at SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, 2005.

[20] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code: Addison-Wesley Professional, 1999.

[21] B. Geppert and F. Rossler, "Effects of refactoring legacy protocol implementations: a case study," presented at Software Metrics, 2004. Proceedings. 10th International Symposium on Software Metrics, 2004.

[22] W. G. Griswold, ``Program Restructuring as an Aid to Software Maintenance," Ph.D. Thesis, Technical Report 91-08-04, Department of Computer Science and Engineering, University of Washington, July 1991.

[23] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting Refactoring Activities Using Histories of Program Modification," IEICE Transactions on Information and Systems, 2006.

[24] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," presented at Software Maintenance, 2002. Proceedings. International Conference on, 2002.

[25] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A Case Study in Refactoring a Legacy Component for Reuse in a Product Line," presented at ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005.

[26] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," presented at Program Comprehension, 2003. 11th IEEE International Workshop on, 2003.

[27] H. Li, C. Reinke, and S. Thompson, "Tool support for refactoring functional programs," presented at Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, 2003.

[28] M. Lippert, Towards a Proper Integration of Large Refactorings in Agile Software Development, 2004.

[29] M. Mäntylä, "An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement," presented at Empirical Software Engineering, 2005. 2005 International Symposium on, 2005.

[30] M, Mäntylä, and C. Lassenius, "Drivers for software refactoring decisions," presented at ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering, 2006.

[31] E. Mealy and P. Strooper, "Evaluating software refactoring tool support," presented at the Australian Software Engineering Conference, 2006.

[32] I. Moore. "Automatic inheritance hierarchy restructuring and method refactoring," In Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (San Jose, California, United States, October 06 - 10). ACM Press, New York, NY, 235-250, 1996.

[33] R. Moser , A. Sillitti , P. Abrahamsson, and G. Succi, Does refactoring improve reusability?, Ninth International Conference on Software Reuse (ICSR-9), Turin, Italy, 11-15 June 2006

[34] G. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?," IEEE Software, 2006.

[35] E. Murphy-Hill. Improving Refactoring with Alternate Program Views. Research Proficiency Exam, Portland State University, Portland, OR, 2006.

[36] A. O`Connor, M. Shonle, and W. Griswold, "Star Diagram with Automated Refactorings for Eclipse," OOPSLA Workshop on Eclipse Technology eXchange, 2005.

[37] W. Opdyke, "Refactoring Object-Oriented Frameworks," 1992.

[38] D. Palvica. “Refactoring Usability Study Report.” http://ui.netbeans.org/docs/hi/javamdr/refactoring-study-report.html, accessed March 2007.

[39] C. Parnin, G, and C. Görg, "Lightweight visualizations for inspecting code smells," presented at SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization, 2006.

[40] B. Pierce, Types and programming languages, MIT Press, Cambridge, MA, 2002.

[41] M. Pizka. “Straightening Spaghetti Code with Refactoring.” In *Proc. of the Int. Conf. on Software Engineering Research and Practice - SERP*, pages 846- 852, Las Vegas, NV, June 2004. CSREA Press.

[42] J. Ratzinger, M. Fischer, and H. Gall, "Improving evolvability through refactoring," presented at MSR '05: Proceedings of the 2005 international workshop on Mining software repositories, 2005.

[43] D. Roberts, “Practical Analysis for Refactoring,” PhD Thesis. University of Illinois at Urbana-Champaign, 1999.

[44] D. Roberts and J. Brant, "Tools for making impossible changes - experiences with a tool for transforming large Smalltalk programs," IEE Proceedings - Software, vol. 151, pp. 49-56, 2004.

[45] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for Smalltalk," Theor. Pract. Object Syst., vol. 3, pp. 253-263, 1997.

[46] B. Shneiderman. Designing the User Interface. Strategies for Effective Human-Computer Interaction (2nd edition) . Reading, Mass.: Addison-Wesley, 1987.

[47] B. Shneiderman, A. Badre, and B. Shneiderman, "System Message Design: Guidelines and Experimental Results," in Directions in Human/Computer Interaction: Ablex, 1982, pp. 55-78.

[48] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics Based Refactoring," presented at CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, 2001.

[49] T. Schrijvers, A. Serebrenik, and B. Demoen, "Refactoring Prolog Code."

[50] J. Shore. The Art of Agile Development. O’Reilly. Under revising as of March 2007.

[51] S. Slinger, "Code Smell Detection in Eclipse." Masters Thesis. 2005.

[52] S. L. Smith and J. N. Mosier. (1986). Design Guidelines for Designing User Interface Software. Technical Report MTR-10090 (August), The MITRE Corporation, Bedford, MA 01730, USA.

[53] M. Streckenbach and G. Snelting, "Refactoring class hierarchies with KABA," presented at OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2004.

[54] L. Tokuda and D. Batory, "Evolving Object-Oriented Designs with Refactorings," presented at ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering, 1999.

[55] E. Van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," presented at WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), 2002.

[56] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards automating source-consistent UML Refactorings," Lecture Note in Computer Science, vol. 2863, pp. 144-158.

[57] M. Verbaere, R. Ettinger, and O. de Moor, "JunGL: a scripting language for refactoring," presented at ICSE '06: Proceeding of the 28th international conference on Software engineering, 2006.

[58] P. Weißgerber, and S. Diehl, "Are refactorings less error-prone than other changes?," presented at MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, 2006.

[59] D. Wells. Refactor Mercilessly. http://www.extremeprogramming.org/rules/refactor.html. Accessed March 6, 2007.

[60] Z. Xing and E. Stroulia, "Refactoring Practice: How it is and How it Should be Supported - An Eclipse Case Study," presented at Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on, 2006.