WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings

Stephen R. Foster
UC San Diego
La Jolla, CA
srfoster@cs.ucsd.edu

William G. Griswold
UC San Diego
La Jolla, CA
wgg@cs.ucsd.edu

Sorin Lerner
UC San Diego
La Jolla, CA
lerner@cs.ucsd.edu

***Abstract*—Integrated Development Environments (IDEs) have come to perform a wide variety of tasks on behalf of the programmer, refactoring being a classic example. These operations have undeniable benefits, yet their large (and growing) number poses a cognitive scalability problem. Our main contribution is WitchDoctor – a system that can detect, on the fly, when a programmer is hand-coding a refactoring. The system can then complete the refactoring in the background and propose it to the user long before the user can complete it. This implies a number of technical challenges. The algorithm must be 1) highly efficient, 2) handle unparseable programs, 3) tolerate the variety of ways programmers may perform a given refactoring, 4) use the IDE's proven and familiar refactoring engine to perform the refactoring, even though the the refactoring has already begun, and 5) support the wide range of refactorings present in modern IDEs. Our techniques for overcoming these challenges are the technical contributions of this paper.**

We evaluate WitchDoctor's design and implementation by simulating over 5,000 refactoring operations across three open-source projects. The simulated user is faster and more efficient than an average human user, yet WitchDoctor can detect more than 90% of refactoring operations as they are being performed – and can complete over a third of refactorings before the simulated user does. All the while, WitchDoctor remains robust in the face of non-parseable programs and unpredictable refactoring scenarios. We also show that WitchDoctor is efficient enough to perform computation on a keystroke-by-keystroke basis, adding an average overhead of only 15 milliseconds per keystroke.

***Keywords*-refactoring; IDE; change detection; repository mining**

I. Introduction

Refactoring is a common activity, yet the automated refactoring support provided by IDEs remains significantly under-used in the wild [1]. This is problematic, considering the tedium of refactoring, the time it can consume, and the considerable possibility of introducing errors. Reasons for the disuse of refactoring have been investigated in depth [2]. Assuming that a programmer has reached a point during development in which a refactoring R is appropriate, there exist several "cognitive preconditions" that must be met before R will be used: The programmer must realize that she is performing a refactoring. She must know that support for R exists. She must know the name of R. She must know that R is applicable. She must believe that invoking IDE support for R is faster than performing R by hand. She must trust that the support for R will not transform her code in unexpected ways. She must be willing to perform a mental and physical context switch from writing code at the keyboard to navigating a menu and GUI wizard via mouse. Moreover, these are merely the cognitive preconditions for *one* refactoring – R – as it pertains to an isolated moment during development. The problem can only become worse with the addition of new IDE-supported refactorings.

We present WitchDoctor, a system that solves the cognitive scalability problem by relieving the programmer of the need to meet the aforementioned cognitive preconditions. WitchDoctor observes the programmer's programming activity, detects when a particular refactoring is in progress, and completes it before the programmer does. In this scenario, the programmer is relieved even of the burden of knowing that a particular refactoring exists – let alone its name, menu location, hotkey, etc.

WitchDoctor's automated recognition can benefit novices and experts alike. Novices are simultaneously struggling with the concepts (e.g., the idea of refactoring), and the environment (Eclipse's refactoring operations). A novice using WitchDoctor can be "taught" in context by having WitchDoctor suggest completions for the novice's stuttering progress. Experts, on the other hand, seek efficiency, and their work can be accelerated if they don't have to interrupt their typing to use the refactoring drop-down menus and its pop-up dialogs. Although the "teaching" interface and the "expert" interface would be quite different, the same underlying technology of WitchDoctor would be required.

Automatically recognizing and completing a refactoring in-progress at the speed of typing poses a number of technical and human-computer interaction challenges. In this paper, we present our solution to five technical challenges that arise when performing "real-time" refactoring detection. In order to be interactive, WitchDoctor must do change detection very quickly; in order to be useful, WitchDoctor must detect and complete the refactoring before the programmer has finished it. These requirements lead to novel technical challenges, where the traditional techniques of change detection cannot be applied straightforwardly:

1) Interactive Speed: We assume that every keystroke that goes unchecked is a missed opportunity to save the

programmer time. Hence, WitchDoctor examines the programmer's document after every keystroke and mouse-click. This requires WitchDoctor to be very efficient in order to avoid noticeable delays while the programmer types. Specifically, it must be much faster than the speed at which programmers can type individual keys while programming. And it must maintain this runtime guarantee even with large files and large projects.

2) Tolerance for Non-parseable Program States: Analyzing the document on every keystroke induces another challenge. In most cases, the document will not be parseable after every keystroke. The programmer will not have terminated the line with a semicolon. The parentheses may not match. There may be statements dangling outside of any method bodies. Indeed, because WitchDoctor must analyze the document after cut-and-paste commands, too, non-trivial and sudden changes to the program's structure may mangle the abstract syntax tree. Thus, WitchDoctor is obliged to treat unparseable programs as first-class citizens.

3) Tolerance for Programmer Variance: The goal of WitchDoctor is to enable a programmer to interface with the IDE's refactoring support regardless of whether she knows it's there. Thus, WitchDoctor must detect refactorings regardless of how they are being performed. Different programmers may carry out the same refactoring in different ways. Take the example of extracting a method. This refactoring is comprised of a number of sub-operations: remove a code block, create a new method declaration, place the removed code block in the method, and introduce a new call to the method. Any permutation of these sub-operations would constitute a method extraction. Taking this a step further, a programmer may even interleave edits not related to the refactoring. WitchDoctor attempts to detect refactorings irrespective of the ordering imposed on its sub-operations by the programmer, as well as allow for unrelated edits in the sequence of edits that includes the refactoring.

4) Reuse of Trusted IDE Components: Our fourth challenge is that we would like to reuse the automated refactoring operations provided by the IDE. Refactoring operations are highly non-trivial to implement. Moreover, experienced programmers are familiar with the "native" operations, and have well-justified expectations for the result of performing a given refactoring. Achieving reuse is non-trivial because – by the time WitchDoctor detects that a refactoring is in progress – the programmer may have already performed multiple sub-operations of the refactoring. The program may or may not be parseable. And even if it is, the state of the program has, by definition, advanced beyond the point where the IDE's built-in refactoring operation ought to have been invoked. The naive solution is to roll the document back to a previous state and then trigger the IDE's refactoring operation. However, this strategy would discard (undo) any unrelated edits that the programmer made between the sub-operations of the refactoring. A smarter roll-back strategy is required.

5) Extensibility: The final challenge is to handle the wide variety of the refactorings supported by an environment like Eclipse – as well as new ones that may be added in the future. WitchDoctor can detect and complete several IDE-supported refactorings – Rename Variable (local and global), Rename Class, Rename Method, Extract Local Variable, and Extract Method. It can also detect and complete one IDE-supported operation which is not a refactoring – Surround with Try/Catch. Further refactorings and even real-time completion support can be added to WitchDoctor with relative ease.

Our techniques for solving these five technical challenges constitute our paper's technical contributions:

- We employ a three-stage strategy of (1) detecting changes, (2) recognizing subsets of changes as the prefix of a refactoring, and (3) rolling back the document to invoke the refactoring tool to create a text view that suggests the refactoring to the programmer.
- To achieve both interactive speed and tolerate unparseable programs, WitchDoctor performs change detection primarily at the text level using Myers's differencing algorithm [3], rather than the AST; a substep maps detected changes to the Eclipse AST.
- To tolerate programmer variance, WitchDoctor matches refactoring specifications against the change history using the Rete pattern-matching algorithm [4].
- To enable reusing Eclipse's refactoring operations, WitchDoctor rolls back just the matched changes (not all the changes since the inception of the refactoring).
- To achieve extensibility, WitchDoctor employs a declarative specification language for describing refactoring operations, akin to Kim's language [5].

In section II, we examine related work as it informs our solutions to the five challenges above. In section III, we give a detailed overview of WitchDoctor. In section IV, we evaluate our techniques for solving the above challenges. In particular, we simulate a highly variant programmer performing more than 5,000 randomized refactoring operations on three Java code bases, finding that WitchDoctor is fast and robust even under these pessimal conditions.

II. Related Work

A. Improved Refactoring Tools

There exists a long history of research on tools for performing refactoring while maintaining program semantics, exemplified by the early work of Opdyke [6] and Griswold [7]. Refactoring support has since been incorporated into numerous modern IDEs – Eclipse, NetBeans, and Visual Studios, for example.

After refactoring became a staple in the toolkit of the modern IDE, the work of Murphy-Hill called into question

the efficacy of refactoring tool interfaces [1], [8], [9]. To our knowledge, the earliest observations pertaining to the cognitive preconditions required by refactorings tools were first made by Murphy-Hill (although he did not use the term "cognitive precondition" *per se*). Additionally, much of the empirical research about refactoring practices in the wild were performed by Murphy-Hill [1], [10], leading to the conclusion that the IDE's refactoring support is underused and that programmers often prefer to refactor by hand.

The same body of work suggests numerous improved interfaces for refactoring tools [2], [9], [11], all of which reduce the cognitive load of the programmer in some way – for example, by assisting in the selection of statements to refactor, by providing more effective context menu support, or by annotating the relevant control-flow and data-flow affected by refactorings. The goal is to make safe, automated refactoring more appealing to programmers who, for whatever reason, often elect to refactor by hand.

Our work draws its motivation from these aforementioned improvements and seeks to further reduce the cognitive preconditions for refactorings. WitchDoctor observes the programmer and attempts to guess whether the user is refactoring by hand. If so, it completes the refactoring. This predictive, on-line refactoring support is a novel way of eliminating the cognitive preconditions even after the user has embarked on the path of refactoring by hand.

B. Predictive Tools

Observing the user and guessing her intent is an oft-employed UI paradigm within IDEs, word processors, and web browsers. It is commonly known as 'completion' – e.g. tab completion, or code completion. Word processors such as OpenOffice will complete a word that the user is typing, drawing from a history of words the user has already typed. Many web browsers, such as Chrome, do the same thing with URLs typed in the search bar, drawing from the user's browsing history or from a list of popular web sites. IDEs such as Eclipse are equipped to complete method names, drawing from the method names in available APIs. In all of these cases, the list of characters already typed by the user becomes a prompt for the environment to provide a completion or list of completions to the user.

WitchDoctor is similar at the metaphorical level. However, instead of observing the sequence of characters typed by the user, it observes the sequence of refactoring sub-operations. And whereas the sequence of characters in, say, a web URL cannot be permuted without altering the URL being typed, the sequence of sub-operations in a refactoring *may* be permuted without implying a change of intent on the part of the user. Furthermore, whereas the completion of a word in a word processor requires observing a sequence of localized changes to the user's document, the completion of a refactoring in an IDE requires observing a sequence of non-localized changes across the user's code base.

C. Refactoring Opportunity Detection

The notion of 'code smells' goes hand-in-hand with refactoring, many smells being placed in correspondence with refactorings that mitigate them [12]. The Long Method smell can be ameliorated by Extract Method. The Large Class can be mitigated by Extract Class. Many tools have been proposed to detect code smells, and hence – to detect opportunities for refactoring [13], [14], [15], [16], [17].

Code smell detection is more analogous to spell-checking than to auto-completion. Code smell detection does not attempt to recognize and extend the user's intent – but rather attempts to direct and modify the user's intent. In particular, such tools draw attention to locations in code for which the user had not formed an intent.

D. Mining Software Repositories

There exists another body of research that attempts to detect refactorings *after* they have happened. This is motivated by the desire to determine how a software framework has evolved, in hopes, for example, of being able to make corresponding evolutions in clients of the framework. Mining software repositories is a process that makes use of high-level differencing techniques to augment the low-level textual differencing techniques of traditional 'diff' tools, accumulating a list of transformations that have affected a software repository [18], [19], [20], [21]. Because these tools have the feature of computing the difference between two consecutive checkins to a version control system, their bar for efficiency is low compared to WitchDoctor. The techniques can also depend on parseable code without much loss of generality.

To summarize, prior work has observed and studied the disuse of refactoring tools in IDEs due to a range of cognitive factors. Whereas code smell detection and framework change analysis attempt to detect refactorings *before* or *after* they happen, WitchDoctor addresses the cognitive factors of disuse by taking the middle ground of attempting to detect (and finish) refactorings *while* they are happening, and in real-time on unparseable programs. We believe this 'in between' space constitutes an untapped research domain with novel benefits, challenges, and techniques in the field of real-time tool support.

III. Design and Implementation

A. Design Overview

WitchDoctor is comprised of three components, each designed to address one or more of the challenges laid out in the introduction. Figure 1 is a graphical overview of WitchDoctor's components. There are three components: *Change Detection*, *Specification Matching*, and *Refactoring Execution*. The architecture can be viewed as a pipeline that successively refines programmer edits into refactorings. During *Change Detection*, programmer edits are analyzed

as changes to the program's abstract syntax tree (AST). During *Specification Matching*, the these AST changes are analyzed to determine if a refactoring is in progress. If so, during *Refactoring Execution*, the refactoring is completed and proposed to the programmer.

Change Detection employs a technique that initially bypasses the AST. This is important to achieve *interactive speed*. But it is also necessary for the purpose of *tolerating non-parseable program states*. Changes are handled first at the textual level, and only later at the syntactical level, ensuring that we do not miss changes due to bad syntax.

Specification Matching involves pattern-matching the programmer's change history against a suite of declarative refactoring specifications. We use an efficient pattern-matching algorithm [4] to avoid re-matching each pattern specification every time a new change is added to the history. The goal of *Specification Matching* is to determine when enough information exists to perform a refactoring on behalf of the programmer. The refactoring specifications are written in a declarative, rule-based syntax that decomposes a refactoring operation into its required sub-operations. This helps meet the challenge of having an *extensible* suite of supported refactorings – making it simpler to define new refactoring specifications. But it also provides a solution to the *programmer variance* problem: the *Specification Matching* process can match a single specification regardless of the order of sub-operations in the programmer's change history and regardless of interleaved changes that do not apply to the refactoring.

Refactoring Execution[1] happens when the *Specification Matching* process has gathered enough information to complete the refactoring. Here, WitchDoctor *reuses trusted IDE components* for the appropriate refactoring. In order to do so, however, the document must be transformed into a state in which the IDE would expect to find it prior to a refactoring. The rollback must also avoid obliterating changes that the programmer interleaved between the sub-operations of a refactoring. *Refactoring Execution* is the slowest step, because WitchDoctor relinquishes control flow to the IDE's refactoring engine, which, although fast, is not designed for *interactive speed*. Thus it is all the more important that *Change Detection*, *Specification Matching*, and the rollback operation of *Refactoring Execution* be as efficient as possible, offsetting the cost of the refactoring engine's invocation.

We now dive into a detailed exposition of these components with the help of a motivating example. Suppose a programmer wishes to extract a code block from one method into a different method that does not yet exist – replacing the code block with a call to the new method. This refactoring is known as Extract Method [12]. Let's say that the programmer's first sub-operation is to highlight the code block and delete it (or "cut" it to the clipboard; the result is the same either way). This operation is visualized in Figure 2.

[1]Our implementation executes the refactoring in the background. A viable alternative would be to prompt the user to trigger the refactoring execution. In this case, "Refactoring Proposal" might precede "Refactoring Execution".

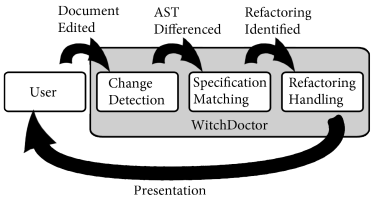

Figure 1. Graphical representation of WitchDoctor's workflow

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

    for(String string : list)
    {
        int string_size=string.length();
        size+=string_size;
    }

    return size;
}
```

Figure 2. A typical delete operation in progress. The user highlights a code block. Deleting the code block triggers the Change Detection phase

B. Detailed Description

1) Change Detection: With the deletion of the statement sequence as shown in Figure 2, the tool begins the *Change Detection* phase.

A tempting strategy for detecting changes to the programmer's program (and one we used in an early version of WitchDoctor) is to parse the document before the change, parse the document after the change, and to use a tool that can compute the difference between the two parsed programs (i.e. UMLDiff [22] or ChangeDistiller [23]). However, this fails to *tolerate non-parseable programs*, and complicates achieving *interactive speed* due to the parsing overhead. The earlier version of the document (call it A), or the later version (call it B), or both versions could contain syntax errors. A syntax error within a method body can cause the entire method body to be missing from the abstract syntax tree.

Our technique is to difference the document at the textual level, and then map the delta to an AST subtree or forest. For text differencing, we use Myers's algorithm [3] – the same algorithm used in the well-known Unix 'diff' tool – because its runtime in practice is very quick when the differences between A and B are small. Because WitchDoctor examines every keystroke and mouse click, A and B often differ

by a single character, making the average runtime of the algorithm proportional to the file size. In section IV, we show that this scales to the largest files seen in practice.

When running Myers's algorithm, newline characters are treated as a delimiter, allowing WitchDoctor to determine line-by-line differences between A and B. WitchDoctor considers contiguous changes to be a single delta: e.g. the deletion of several consecutive lines is considered to be a single deletion operation – one that may involve more than one AST node. After running Myers's algorithm, WitchDoctor then attempts to produce the original and revised ASTs – AST(A) and AST(B) – and map the textual differences to relevant AST nodes. To do so, WitchDoctor attempts to find the set of AST nodes that the textual change "covers":

$$\{n \quad | \quad \textit{offset}(n) \geq \textit{offset}(change) \wedge \textit{length}(n) \leq \textit{length}(change) \wedge n \in \textit{ASTFor}(change)\} \tag{1}$$

where $\textit{offset}(n)$ is the textual offset of entity n from the beginning of the document, and $\textit{length}(n)$ is the textual length. The function $\textit{ASTFor}(change)$ is shorthand for the table below. For a given textual difference, we have various editing possibilities: it could be an insertion, a deletion, or an update (effectively an atomic delete/insert at the same textual offset). The type of the change determines which AST is used by WitchDoctor when mapping the change. The following table summarizes which AST is used in each case:

	AST(A)	AST(B)
Insert		Yes
Delete	Yes	
Update	Yes	Yes

The special case is Update, for which both the before (from file A) and after (from file B) ASTs are mapped.

Returning to our example, we observe that the removal of several lines from a method will register as a Delete operation after running Myers's algorithm. According to the table, we select file A to look for the AST nodes that map to the textual change. From file A's AST, we find the set of statement nodes that are covered by the textual changes – using Equation 1.[2]

Our approach is *tolerant of unparseable programs* through a multi-step strategy. First, In the case of insertions, any syntax errors in file A are irrelevant. Likewise, in the case of deletions, any syntax errors in file B are irrelevant. Thus, the document may move from a parseable state to a non-parseable state, or vice versa, and WitchDoctor can often still find the AST nodes it is looking for.

Even when a file is not parseable, all hope is not lost. Eclipse's built-in AST parser is quite forgiving and will parse as deeply along each AST branch as possible. A syntax error in a method m may cause m's branch to be non-parseable. But if the relevant textual change pertains to a different method, then WitchDoctor can still find a mapping using the textual change's character offset according to Equation 1.

The remaining, difficult case is when the textual change *cannot* be mapped to an AST node – due to syntax errors (1) in the relevant file, and (2) in the same branch affected by the textual change.

The first strategy is to attempt to construct an AST out of the textual change itself. Often, programmers cut-and-paste blocks of code that can be parsed as a sequence of statements. Even when the AST of the document is mangled beyond repair – sometimes the textual delta itself can be used to construct a smaller AST. In the case of the deleted statements in our example, even if the rest of the document was not parseable, the removed statements themselves would be parseable.

The second strategy is to find a relevant ancestor AST node – an AST node whose associated character offset and length indicate that the textual change is "covered by" the AST node. Of the covering nodes

$$\{n \quad | \quad \textit{offset}(n) \leq \textit{offset}(change) \wedge \textit{length}(n) \geq \textit{length}(change) \wedge n \in \textit{ASTFor}(change)\} \tag{2}$$

we select the one whose length is the smallest. The upshot is that, even if WitchDoctor cannot figure out an exact mapping to an AST node, it can often find a parent AST node that was affected by the change. This can be useful, for example, if the programmer pastes a block of statements into the space between methods of a class – a sub-operation that can signal that the programmer is performing Extract Method and plans to immediately wrap the statements inside a method body. The covering node, in this case, is the class. Although the pasted statements will cause the document to become largely non-parseable, the affected class can still be obtained – allowing WitchDoctor to determine the target class for the Extract Method operation.

The goal of *Change Detection* is, after all, to detect as many relevant changes as possible – giving the *Specification Matching* phase enough information to detect refactorings in progress. Performing the textual differencing algorithm first, followed by a mapping to AST nodes allows information to be gathered even when the program is highly non-parseable.

[2]WitchDoctor does some optimization here in service of *interactive speed*. Because the *Change Detection* phase occurs after any change to the document, WitchDoctor keeps the computed AST(B) in memory, using it as AST(A) in the next *Change Detection* phase.

Continuing with our example, the deletion of statements from a method has, during *Change Detection*, been detected by Myers's algorithm and mapped to an appropriate AST node (and by indirection its subtree). We structure the result as a tuple:

$$(delta\_type, ast\_nodes)$$

where ast_nodes is the set of AST nodes to which the textual delta was mapped during *Change Detection*, and

$$\begin{aligned} delta\_type \in \quad & \{Insert, Delete, Update\} \times \\ & \{Covers, CoveredBy\} \end{aligned}$$

The tuple is then passed to the *Specification Matching* component. Because all but the most exceptional of our AST-mapping techniques outlined above produces an AST node that the textual change *covers*, most tuples encountered in WitchDoctor's pipeline will have a $delta\_type \in \{Insert, Delete, Update\} \times \{Covers\}$. To simplify the following exposition, then, we will assume that the textual change covers the AST node unless stated otherwise, writing it as $delta\_type \in \{Insert, Delete, Update\}$.

2) Specification Matching: In this phase, the sequence tuples created by the *Change Detection* phase – one for every edit – are pattern-matched to detect evidence of refactorings. A naive, slow approach would be to keep the tuples in an edit history and to search for patterns within the entire history each time a new tuple is added. Instead, WitchDoctor employs the Rete algorithm [4] – an algorithm for efficient pattern-matching in production-rule systems. We keep a list of refactoring "specifications" (patterns) that represent refactorings that may be in progress. Each new tuple is checked against the list of specifications. The specifications can be thought of as state machines: If the new tuple is evidence of a refactoring that the specification is designed to recognize, the specification's internal state advances. As a result, each new tuple can be evaluated just once, obviating the need to sift through an ever-growing history of changes looking for matches.

The specifications are designed using a declarative rule-based model, derived from Kim's body of research on detecting refactorings in software repositories [21], [5]. Each refactoring is described by one or more "specifications". For example, here is a simplified specification to detect a subset of the ways in which Extract Method may be performed:

$$\begin{aligned} & (Delete, code\_block) \\ \wedge \quad & (Insert, method\_call) \\ \wedge \quad & (Insert, new\_method) \end{aligned}$$

Each constituent of the conjunction describes a change tuple: giving the change type and a descriptor for the AST node set. The descriptor can be arbitrarily complex – and is implemented as an AST node visitor, which can traverse ASTs both upward and downward, scouring the tree for various properties. However, in practice, we have found it sufficient to implement these AST node set descriptors as simple checks against the types of the AST nodes. For example, the $code\_block$ descriptor checks for a sequence of statements, and the $method\_call$ descriptor checks for a method call AST node.

This is too simple, however. One can imagine scenarios in which a programmer deletes a code block and adds a method – but is not performing a refactoring. The heavy-lifting in WitchDoctor is done by the *constraints* attached to each specification. For example, we could flesh out the above specification with constraints as follows:

$$\begin{aligned} & (Delete, code\_block) \\ \wedge \quad & (Insert, method\_call) \\ \wedge \quad & (Insert, new\_method) \end{aligned}$$

$$where$$

$$\begin{aligned} & position(code\_block) = position(method\_call) \\ \wedge \quad & name(method\_call) = name(new\_method) \end{aligned}$$

Of course, this is an over-simplification because the $position$ function must take into account the *deleted* position of $code\_block$ versus the *added* position of $method\_call$. Furthermore, the ASTs for $code\_block$ and $method\_call$ may look very different – so a simple calculation of the offset is not usually an adequate way to determine the position. Instead, properties of the ASTs must be used to determine whether the positions are equal.

Although this high-level presentation obscures some of the implementation details, it illustrates the distinction between the components of a specification, *requirements* and *constraints*. The requirements can be thought of as patterns that must be matched. The constraints relate the requirements to each other, preventing some requirements from matching. Each requirement that is matched can be thought of as the advancement of the specification's state machine.

Continuing with our example, the deleted code block will be matched by the requirement $(Delete, code\_block)$. The first constraint is not applied because $position(method\_call)$ is undefined (another subtlety, but easily implemented); the second constraint is ignored because it does not pertain to the code block. When the matching occurs, the $(Delete, code\_block)$ requirement establishes a binding with the tuple generated during *Change Detection*. A specification begins with no bindings and gradually acquires bindings until they carry enough information to execute the refactoring (i.e., supply

all the required parameters to the Eclipse refactoring).[3] Thus, in our example, the specification shown above has now established one binding.

Whenever a specification is updated, the original is always retained – unchanged. In our example, the specification for Extract Method now has a binding for $code\_block$. However, the original Extract Method specification remains in the list of specifications. The point is that, if the programmer deletes another code bock, the specification whose $code\_block$ is bound will ignore the change, but the specification whose $code\_block$ is unbound will acquire a binding. This allows WitchDoctor to detect the advent of a newly begun refactoring regardless of which refactorings it believes to be in progress. Indeed, many refactorings are permitted to be in progress at once. For example, a programmer may begin extracting one method, proceed to extract a completely different method, and return to complete the original refactoring. This is one form of *programmer variance* that WitchDoctor permits.

The challenge of supporting a *broad and extensible* suite of refactorings is partially accomplished through the use of these specifications, which cleanly model the requirements for a refactoring. Requirements and constraints, once implemented as Java classes, may be reused between specifications.

The challenge of *tolerating programmer variance* is accomplished in several ways using these specifications. For one, the requirements for a specifications may be matched in any order, permitting the programmer to permute the sub-operations defined by the requirements. Furthermore, the programmer may interleave edits between the sub-operations of a specification. They will not affect the state of any specifications unless they are relevant to a refactoring.

However, for the sake of efficiency, the list of in-progresss specifications cannot be allowed to grow indefinitely. For the sake of efficiency, WitchDoctor removes specifications whose states have not progressed recently. Consequently, if the programmer interleaves enough edits between relevant sub-operations, the refactoring may be missed due to the excessive noise.

Continuing our example, suppose the programmer now places a method call into the method where the code block was deleted. Figure 3 visualizes this insertion. The *Change Detection* process will run, creating the relevant change tuple. The inserted method call matches the second requirement in our specification $(Insert, method\_call)$, and the constraint is satisfied. So the binding takes place, advancing the state of the specification. With two requirements bound, the specification has enough information to trigger WitchDoctor's next phase. WitchDoctor now knows the

[3]In some cases, a binding can be supplied automatically by WitchDoctor, rather than supplied through programmer edits, such as providing a default legal name for a method. This name can be changed later by the programmer.

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

    for(String string : list)
    {
        findSize();
    }

    return size;
}
```

Figure 3. An insert operation, triggering the Change Detection and Specification Matching phases for the second time – also providing enough information to trigger the Refactoring Execution phase.

contents of the code block being extracted and the name of the method into which the block will be extracted. This is enough information to execute the refactoring.

3) Refactoring Execution: The goal of this phase is to complete the refactoring that has been detected. Once completed, the refactoring can be proposed to the programmer.

The challenge is that, on the one hand, we would like to *reuse well-tested components* in the IDE; yet on the other hand, the refactoring has already been initiated by the programmer. Thus, the document will not be in the state that the well-tested components expect it to be in. Some of the programmer's changes need to be reverted in order to return the document to a state in which the refactoring has not yet begun.

Unfortunately, a general approach cannot work – such as restoring the document to its state before the first sub-operation of the refactoring. The edits interleaved between sub-operations would be lost.

Instead, the edits matching the specification must be rolled back while preserving the interleaved edits. In the case of our example, the rollback strategy is to inline the code block, replacing the method call. In essence, we run the refactoring backward, independent of the interleaved edits. Then, we call refactoring engine – in the background – to perform Extract Method, feeding it the code block and the name in the method call.

During this phase, WitchDoctor must relinquish control flow to the IDE's refactoring engine. The refactoring engine's operations are tuned for correctness, not speed. So this is the one phase of WitchDoctor in which the programmer might detect a noticeable delay in typing, although the refactoring is happening in the background, minimizing distractions to the programmer. In our evaluation, we assess how long this delay is in practice.

Continuing with our example, the document has been rolled back and the refactoring has been performed in the background, yielding a version of the programmer's document in which the refactoring has been fully performed. There are a variety of ways to present the results of the

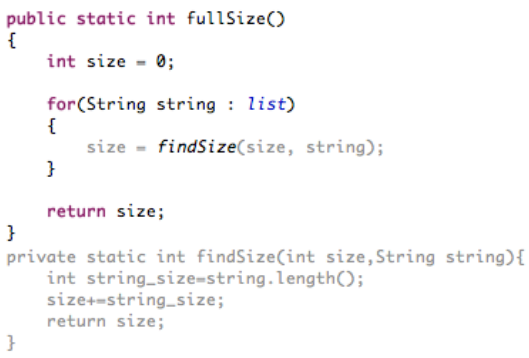

Figure 4. The tool's proposal, displaying proposed changes in gray.

refactoring, which is not a consideration for this paper. An informal poll of our colleagues has turned up a myriad of strategies for proposing the completed refactoring to the programmer. These various user interface options and their relative merits are beyond the scope of this paper and will be compared and tested in future work. Regardless of the presentation, efficiency is still a prime consideration. For now, we describe the interface that we currently use for testing WitchDoctor.

In our running example, the execution of the refactoring engine yields a document in which the inserted method call (which was inlined during the roll back) has now been reinserted by the refactoring engine, and the code block the programmer deleted is now wrapped in a method declaration. Our interface proposes this change by displaying the new method declaration in gray, to indicate that it is a suggestion. The rest of the code (everything typed by the programmer so far) is rendered in normal colors (see Figure 4). At this point, the behavior mimics tab-completion. If the programmer presses "Enter" or "Tab", the changes are finalized. If the user continues typing in such a way that doesn't contradict the suggested change, the suggestion remains in effect. Otherwise, the suggested changes are removed.

IV. Evaluation

A. Experimental Methodology

Key experimental questions revolve around the speed of our approach, its ability to handle unparseable programs, and its ability to handle variance in the way programmers can perform refactorings by hand. To gain insight on these, performing a large number of complex refactorings in a great variety of ways on a variety of software projects is appropriate.

Refactoring Type: We chose to focus on Extract Method, because it is amongst the most complex available in Eclipse, and the most complex that we implemented. It comprises multiple steps and has elements that are related to other, simpler, refactorings such as rename and extract binding. Also, there are numerous ways to perform Extract Method, many of which create non-parseable states.

Simulated User: Ideally, we would assess WitchDoctor's design and implementation against thousands of by-hand refactorings, based on the techniques (styles) of real programmers, with a validated distribution of the application of those techniques. To our knowledge, these don't exist.

Instead, we chose to simulate the broad spectrum of possible Extract Method refactorings in a randomized, unbiased distribution. Although not externally valid, a random distribution still allows examining selected subpopulations of the refactoring pool and discussing their likely relation to practice.

To this end, we built a simulated user that can make scripted changes to a document. We programmed the user to randomly select, construct a script for, and perform 5,000 correct Extract Method refactorings. Each editing operation in each script is executed by the simulated user by mimicking human behaviors such as keystrokes, mouse clicks, selections, cut operations, and paste operations. The simulated user ran at a rate of 50 milliseconds per keystroke – much faster than the average human typist.

As part of the simulated user's "random" behavior, it often "forgets" to close curly braces or to add semi-colons. It also inserts lines character-by-character, causing the line to be non-parseable until the final semi-colon. When deleting a sequence of lines, it deletes them in a random order, which often results in non-parseability – e.g. when curly braces or control constructs are deleted.

Additionally, to incorporate the notion of user variance into our simulated programmer, it randomized the order of the sub-operations used to perform each Extract Method refactoring. Sometimes, the new method declaration would be added first, followed by the deletion of the code block and addition of the method call. At other times, the order was permuted. The order was randomly chosen for each script, yielding a uniform distribution across the various permutations. Pseudo code for simulating a single refactoring might look like:

```
let ops = the refactoring sub operations
scramble_order_of(ops)

for each op in ops
  switch(rand)
    case 1:
      do_keystrokes(op)
    case 2:
      do_cut_and_paste(op)
    case 3:
      do_keystrokes_but_no_curly(op)
    case 4:
      ...
end
```

	Files	Refactorings Performed
Eclipse Compare Plugin	157	1421
Eclipse JFace Plugin	381	1330
Apache Struts	1932	2407

Figure 5. Projects tested, with the total number of files in the project and the number of random refactorings run by our simulated programmer.

Projects: We chose three open-source Java projects: Eclipse Compare Plugin, Eclipse JFace Plugin, and Apache Struts. These projects are somewhat dissimilar: they perform rather different tasks, and were implemented by different teams. Figure 5 gives a profile of the three projects. Although these projects are not especially large project size did not turn out to influence performance.

Execution Environment: Finally, our experiments were run on a MacBook Pro, with 4G RAM and 2.4 GHz Intel Core 2 Duo processor. Such a machine, while contemporary, is modest in its clock rate and number of cores. The operating system is MacOS 10.6.7. All refactorings ran in Eclipse 3.6.1 (Helios).

B. Results

1) Tolerance of Non-parseable Program States and User Variance: WitchDoctor was able to detect the refactoring performed by the simulated user 89% to 93% of the time. WitchDoctor was also able to complete the simulated user's intended refactoring before the user could complete it 33% to 46% of the time (See Figure 6).

Extreme user variability was the primary cause of WitchDoctor's failure to complete a refactoring, in particular order of the suboperations. One such order involves first adding the new method declaration (complete with the body) before adding the method call or deleting the code block. In these cases, WitchDoctor detects the new method declaration but does not complete a refactoring because it cannot determine the code bock to extract. If the programmer then adds a call to the new method, WitchDoctor still does not know the code block to extract. By the time the code block has been removed, all of the work of the refactoring has already been performed – leaving no time to complete the refactoring on behalf of the user. A similar failure can occur when the method call is added first, followed by the method declaration (with body), and the removal of the code block.

Half of WitchDoctor's failures to complete the refactoring can be attributed to these late detections – after which there is no work to perform. Yet, it seems unlikely for programmers in the wild to construct a method declaration complete with its final body as the first step of Extract Method. This technique requires significantly more work than utilizing cut-and-paste – e.g. cut the code block, paste it into a new method, add a method call. Omitting just these twp obviously anomalous cases would raise the completion rate to about 2/3.

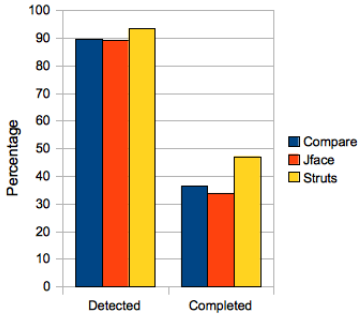

Figure 6. WitchDoctor's percent of refactorings detected and percent of refactorings completed on each of the three experiments.

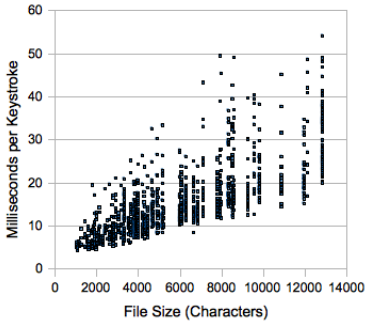

Figure 7. WitchDoctor's average overhead in keystrokes, scaling linearly with the size of the file. The chart summarizes the experment run on the Eclipse Compare plugin.

2) Interactive Speed: The typical typist is said to type at 300 characters per minute (http://en.wikipedia.org/wiki/Words_per_minute). This equates to 200 milliseconds between each keystroke. In Figure 7, each data point represents the average computation per keystroke within a particular refactoring performed by the simulated user during the experiment run on the Eclipse Compare plugin. WitchDoctor performed even better on the JFace and Struts experiments. The chart show that the computation performed by WitchDoctor after each keystroke is, on average an order of magnitude less than 200. The time to process a keystroke was never more than 62.2 milliseconds (this includes rollbacks). We do note however, that processing time increases linearly with respect to file with a slope of 1/3.

3) Reuse of Trusted IDE Components: In all cases, the completion of the refactoring was performed by invoking Eclipse's refactoring tools – which requires the selective rollback process described previously. Surrendering control flow to the well-tested Eclipse refactoring engine constitutes the most significant impact to WitchDoctor's performance.

The following table summarizes the averages across the three experiments.

	WitchDoctor	Eclipse Refactoring
Compare	15 ms	974 ms
JFace	12 ms	856 ms
Struts	7 ms	848 ms

The WitchDoctor column is the average time WitchDoctor spends computing per keystroke across the three experiments (a short summary of the data presented in 7). The Eclipse Refactoring column is the average time taken for Eclipse to complete refactorings detected by WitchDoctor.

In all cases, the refactoring takes significantly more time than the amount of time spent within WitchDoctor. Although this delay is significant, we feel that using the trusted components of the IDE are justified for three reasons not just for robustness and familiarity, but also because: (1) the noticeable delay cause by the refactoring engine is not the normal case for WitchDoctor, which still performs its normal computations in less than 15 milliseconds on average, and (2) the delay caused by the refactoring engine is a prelude to a completed refactoring, which saves the user significantly more time in the long run.

We would like to add that Eclipse's refactoring implementation cannot be "blamed" for being slow. It is only in the context of real-time support tools like WitchDoctor that the notion of *interactive speed* becomes an issue for the IDE's refactoring engine. If real-time interactive refactoring becomes common, optimizations for refactoring engines will follow.

4) Extensibility: We have specified (implemented) 6 refactorings in WitchDoctor: Rename Variable (local and gobal), Rename Type, Rename Method, Extract Local Variable, Extract Method, and Surround with Try/Catch (a non-refactoring operation supported by Eclipse). In all cases, the specification technique we used to define refactorings allowed us to incorporate these new refactorings with relative ease, giving us confidence that the suite of refactoring operations in Eclipse can be supported by WitchDoctor as well as new refactoring operations that may be added to Eclipse in the future. For example, it was fairly trivial to implement the specification for Rename Variable after having implemented Extract Method. In the following sample implementation, the classes Delete, Insert, NotEqual, and SamePlace are all reused from Extract Method:

```
Specification rename =
  new Specification("Rename Method");
rename.addRequirement(
  new Delete("old_name"));
rename.addRequirement(
  new Insert("old_name"));
rename.addConstraint(
  new NotEQ("old_name","new_name"));
rename.addConstraint(
  new PlaceEQ("old_name","new_name"));
```

V. Future Work

Having models of how programmers refactor by hand can inform the development of tools like WitchDoctor, and may have other applications to informing software engineering practice. We plan to deploy WitchDoctor in an "observation" mode that will record all the ways that users refactor by hand. This approach will permit gathering a very large and detailed "in the wild" dataset.

A key question in this line of research is the efficacy of real-time refactoring suggestions. Given that the performance and robustness of WitchDoctor is high, the main outstanding design question is interface design. An informal poll of our colleagues suggests a potential need for a wide variety of different interfaces for proposing refactoring completions to human programmers – from actually transforming the document under the nose of the programmer, to various kinds of pop-up context menus, to code annotations, to a full-fledged video tutorial for novices on the best practices and merits of the refactoring being performed. We plan to prototype interfaces for novices and experts, ultimately performing lab and field studies of the use of WitchDoctor and the practices that arise around it.

VI. Conclusion

Refactoring tools are underutilized due to the numerous cognitive preconditions that must be satisfied before a refactoring operation is invoked. WitchDoctor represents a first step toward eliminating these preconditions. A programmer can simply write code as she normally would, and WitchDoctor makes refactoring suggestions. We have shown that the techniques applied in WitchDoctor achieve very complete, extremely fast recognition of refactorings. Making assumptions about how real programmers refactor, the suggestion rate is high as well. Key future work is to investigate how programmers refactor by hand in the wild, and to design and investigate user interface designs for real-time refactoring suggestions.

Several techniques were required to achieve fast, robust, and extensible refactoring suggestions. Textual differencing can be combined with delayed AST node mapping to handle non-parseable program states. A declarative specification language and associated pattern-matching algorithm can be used to recognize a broad and extensible array of refactorings as they unfold, regardless of user variance. Selective rollback permits invoking the IDE's refactoring engine while tolerating interleaved edits.

Acknowledgements

This research supported in part by an NSF Graduate Fellowship and a Google Research Award.

REFERENCES

[1] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–297. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070529

[2] E. Murphy-Hill and A. P. Black, "High velocity refactorings in eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, ser. eclipse '07. New York, NY, USA: ACM, 2007, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1328279.1328280

[3] E. W. Myers, "An o(nd) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[4] C. L. Forgy, "Expert systems," P. G. Raeth, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, ch. Rete: a fast algorithm for the many pattern/many object pattern match problem, pp. 324–341. [Online]. Available: http://dl.acm.org/citation.cfm?id=115710.115736

[5] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 371–372. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882353

[6] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[7] W. G. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, 1991.

[8] E. Murphy-hill, "Improving refactoring with alternate program views. research proficiency exam," 2006.

[9] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: observations and tools for extract method," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 421–430. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368146

[10] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin, "Gathering refactoring data: a comparison of four methods," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT '08. New York, NY, USA: ACM, 2008, pp. 7:1–7:5. [Online]. Available: http://doi.acm.org/10.1145/1636642.1636649

[11] E. Murphy-Hill, "Improving usability of refactoring tools," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 746–747. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176705

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[13] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 2002, pp. 97 – 106.

[14] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *In Proc. IEEE International Conference on Software Maintenance*, 2004.

[15] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the 11th IEEE International Software Metrics Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 15–. [Online]. Available: http://dl.acm.org/citation.cfm?id=1090955.1092155

[16] N. Moha, Y.-G. Guhneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells." *IEEE Trans. Software Eng.*, pp. 20–36, 2010.

[17] I. M. Bertran, "Detecting architecturally-relevant code smells in evolving software systems," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 1090–1093. [Online]. Available: http://doi.acm.org/10.1145/1985793.1986003

[18] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP*, 2006, pp. 404–428.

[19] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 818–836, December 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1314032.1314041

[20] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 325–334. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806848

[21] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, sept. 2010, pp. 1 –10.

[22] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 54–65. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101919

[23] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *Software Engineering, IEEE Transactions on*, vol. 33, no. 11, pp. 725 –743, nov. 2007.