Immediate Search in the IDE as an Example of Socio-Technical Congruence in Search-Driven Development

Adrian Kuhn
Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch/akuhn

ABSTRACT

Search-driven development is mainly concerned with code reuse but also with code navigation and debugging. In this essay we look at search-driven navigation in the IDE. We consider Smalltalk-80 as an example of a programming system with search-driven navigation capabilities and explore its human factors. We present how immediate search results lead to a user experience of code browsing rather than one of waiting for and clicking through search results. We explore the socio-technical congruence of immediate search, *i.e.* unification of tasks and breakpoints with method calls, which leads to simpler and more extensible development tools. Eventually we conclude with remarks on the socio-technical congruence of search-driven development.

1. INTRODUCTION

Search-driven development aims to satisfy the information needs of developers through novel technical solutions. Search-satisfiable information needs have been observed during (at least) three different activities: code reuse, code navigation, and program debugging. Common to most of these activities is that they are dominated by *both* human factors and technical issues. For example, in order to reuse external source code, developers have to trust the external code (trustability, a human factor) as well as to adapt the external code to the their local source base (suitability, a technical issue).

In this essay we look into the socio-technical issues of search-driven navigation in the IDE. We consider Smalltalk-80 as an example of a programming system with search-driven navigation and explore its human factors. As the Smalltalk language was designed *together* with the Smalltalk IDE, we have the case of a language that was shipped from the first day together with an IDE that supports immediate search. This led to interesting best practices.

For example, it is common practice to mark pending tasks by calling a method named `#todo` (which is defined on Object and thus valid in all source code) rather than putting the term `todo` in a comment. In order to view all pending issues developers will just search for all references to the `Object#todo` method—and since the result opens immediately, the developer will refer to this as "browsing all todos." This leads to the unification of otherwise unrelated concepts such as tasks and even breakpoints(see Subsection 3.3) with simple method calls. As a direct technical benefit of this unification design and implementation of the IDE become simpler. As an indirect benefit developers can add new concepts (*e.g.* a new kind of task or breakpoint) in the same way they add methods to the system, there is no need to learn the arcane skills of writing an IDE or VM plug-in.

The remainder of this essay is structured as follows: Section 2 provides a general introduction to search-driven development in order to clarify the context of this paper. Section 3 considers Smalltalk-80 as an example to explore socio-technical issues of search-driven code navigation (parts of which extend upon a recent blog post by the same author[1]). Eventually, Section 4 concludes with a summary and remarks on research challenges.

2. SEARCH-DRIVEN DEVELOPMENT

To our best knowledge, no taxonomy of search-driven development has been proposed yet. It seems to be common though to define search-driven development in terms of those information needs of developers [12] that can be satisfied through search (including both active and proactive search (*e.g.* [21, 9]), *i.e.* search queries with *pull* and *push* based result streams as well as with *automatic* and *human* query formulation).

Information needs that can be satisfied with search have been observed as part of (at least) three different activities: code reuse, code navigation, and program debugging. The information needs of code reuse are satisfied by searching external sources, such as the internet or a local code repository. The information needs of code navigation are typically satisfied within the local code base through the means of an IDE (integrated development environment). The information needs of program debugging have received little attention by the SUITE community so far; some of them are satisfied by searching back-in-time through the program state (*e.g.* [13, 15]).

In the remainder of this section we will discuss search-driven reuse and search-driven navigation in more detail, and eventually end with remarks on search user interfaces.

[1] http://www.iam.unibe.ch/~akuhn/blog/2010/imagine-ide-search-so-faaaaast-that/

SUITE '10, May 1 2010, Cape Town, South Africa

The promise of *search-driven reuse* is that developers will locate pieces of (possibly external) source code through the means of search and reuse the search results in their local code base [11]. This is based on observations of the actual behavior of software developers and end-user programmers, in fact current internet search engines are often used to locate both snippets of code as well as complete components. It has been found that, to enable their reuse, not only the relevance but also *trustability* and *suitability* of search results are important [20]. In general trustability is a human factor and suitability is a technical issue. But there are socio-technical congruences too, *e.g.* Steven Reiss conjectured that developers might be less likely to reuse a search result if it does not adhere to their personal coding convention and style guide [16].

In the IDE, search is a useful means of navigation. We refer to search that is embedded in the navigation features of the IDE as *search-driven navigation*. For example, identifiers names that are turned into hyperlinks to their definition. We distinguish this from search that is separate from navigation and often accessible through a complex dialog only; but also from drill-down navigation that is not search based, as *e.g.* in a hierarchical code outline.

The promise of *search-driven navigation* is that developers will save development time that would otherwise have been spent with searching the local code either manually or with primitive search tools (*i.e.* keyword search, text-based regular expressions). This is based on observations that developers spend most of their time in the IDE with navigation and search [18] and that software developers are often lost in code [2]. Developers can be lost in two ways: either they do know the cognitive clue (*e.g.* method name, or its functionality such as concatenating strings) of their target but not its precise location in the source code, or they do not know at all where to go next (*e.g.* when starting to work on a bug report).

A major short-coming of most IDE-based search is the search user interface [19]. A good search interface should be simple and lucid [8]. Search results are often presented in IDE elements that parallel the normal navigation means, *i.e.* outline and editor. If we compare this with *e.g.* compiler warnings that are typically embedded in outline and editor, then there is clearly room for improvement. The same holds for query formulation, which is typically a modal dialog with a single-line text field and a myriad of check boxes and radio buttons. It has been suggested to replace this dialog with at a text area [10], the current context or selection in the source code [9], or even to use unit tests as search queries [14, 11].

3. EXAMPLE: SMALLTALK-80

In this section we consider Smalltalk-80, an early object-oriented programming language and development environment. For this essay we selected[2] Smalltalk-80 because it is one of the few industry-proof systems where programming language and development environment have been created together, which has led to interesting best practices. Also Smalltalk-80 is different enough from today's mainstream systems for these practices to be evident even without conducting a user study. Smalltalk pioneered the use of graphical user interfaces at a time where there were no modern operating system, thus both programming language and IDE had been developer by the same team at the same time. In the 80ies and 90ies, Smalltalk was widely used in both industry and academia.

3.1 Technical Details

In this subsection we present just enough technical detail of Smalltalk in order to understand and discuss the issues that relate to search-driven development. For a full coverage of Smalltalk, please refer to either the original documentation [7, 5, 6] or recent tutorials [1].

Smalltalk pre-dates modern operating systems with their hierarchical file systems and has thus some rather uncommon features. Of interest with regard to code search is how the sources of a Smalltalk are handled. A Smalltalk system consists of two parts, the "image" which is a core dump of the running system and the "sources file" which is a disk-based cache for source code. The memory dump contains all objects, that is both all domain objects of the running application and the full class model of the system (including the byte-code of all method, but not their source code). The sources file contains all source code of the system,

Smalltalk does not distinguish between development and deployment: all development is done on a running system, updating classes and methods at runtime (*i.e.* hot-swapping of code). All code is always compiled and available as byte-code, even during development[3]. In the byte-code of Smalltalk method, the names of all called methods are stored as symbols (*i.e.* interned strings) in a literal array. Therefore, searching for all references to a method is a very fast operation, which allows the IDE to immediately open the search result, without having to go through the performance bottle neck of parsing all source code and resolving all bindings.

In a Smalltalk-80 system, the main means of navigating is to search for all references or declarations of the target method's signature (this is done by pressing Ctrl-m or Ctrl-n on a method name). The results of this search pop up immediately, and therefore developers refer to this search as "code browsing" since their user experience does not include any observable search activity. The same is the case for *e.g.* compilation: if compilation is immediate, then (from the point of the user experience) compilation ceases be experienced and eventually becomes equivalent to "just running the code".

3.2 Search Capabilities

Figure 1 illustrates all search capabilities of Smalltalk that are presented below (from left to right): browse senders and implementers, cascading code browser with search bar, the code rewrite panel, and the method finder tool.

In the Smalltalk IDE the actions "browse senders of" and

[2]If I my allow myself a personal note—of course, as is not uncommon in science, causality of the actual scientific conduct is reversed to the one presented in the paper. My occupation with Smalltalk predates my interest in search-driven development. I am with the Software Composition Group, which makes heavy use of Smalltalk to explore new programming languages features, and it was in that context that I became aware of the socio-technical impact of Smalltalk's immediate search. And even then, I was not aware of it unless my advisor asked in a private tweet: "How do you flag to-do items in Java? Annotations or dedicated methods?"

[3]In fact, Eclipse got its ability to keep all code compiled at all times from its predecessor VisualAge which was IBM's prime Smalltalk IDE before they switched from Smalltalk to Java.

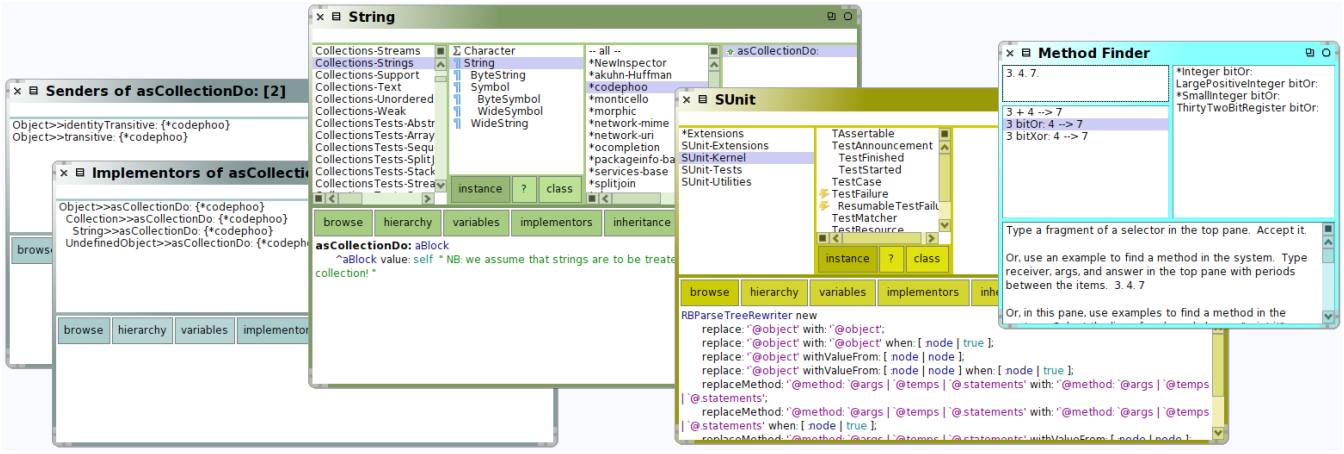

Figure 1: All search capabilities that are presented in Section 3 (from left to right): browser senders and implementers, cascading code browser, the code rewrite panel, the method finder tool, and on top of all windows (except the method finder tool on the left) the code browser's search bar.

"browse implementers of" are the main means of navigation. Executing these actions opens – immediately – a new editor window with all callers (respectively implementers) of a given method. The search queries find all references (respectively declarations) of a method and immediately open all results in a code editor. Of course, not all navigation needs can be satisfied by hyper-jumping. Sometimes developers need to drill down from top-levels packages to methods. To do so Smalltalk offers a tabbed code browser which is the predecessor of Eclipse code browsing perspective.

Over the years, more capabilities were added: the code rewrite panel of the refactory browser, the method finder tool, and the code browser's search bar.

Smalltalk's REFACTORY BROWSER (RB) was the first IDE with support for automated refactorings [17], and it got a feature that is, to our best knowledge, still unique. Internally, the RB uses an extension of Smalltalk's abstract syntax tree to search (and replace) the abstract syntax tree of source code. The same syntax is also exposed to the developers through a code rewrite panel. In the code rewrite panel developers can enter Smalltalk source code – with special wildcards for pattern matching – in order to search for (an possibly rewrite, using search and replace) any kind of code snippet.

Alas, the code search of RB is rather slow. In order to search all ASTs, the RB must (same as *e.g.* Eclipse for a simple references search) parse all source code of the system, which is slow. However, in the context of sub-method reflection it has been showed that it is feasible to store ASTs rather than byte-code in the Smalltalk image [3], that would bring RB's code rewriting up to almost the same immediate speed as Smalltalk's browsing of callers and senders.

The METHOD FINDER (MF) tool allows developers to find unknown methods. The developer must not know the name of the methods. MF expects a list of parameters (which includes the receiver) plus an expected return value. MF will iterate through all permutations of the expected parameters, calling all available methods of the permutation's first element, and eventually returns all methods that returned the expected return value. MF keeps a black list of possibly harmful methods.

The most recent addition to the Smalltalk IDE is the MERCURY search bar. The bar fills the full width at the top of each code editor window. It can be used to search for references, declarations, strings in string literals (which is fast as it searches the running "image" file) as well as full-text search (which is slow as it searches the external "sources file" cache).

3.3 Socio-technical issues

Since the Smalltalk language was designed *together* with the Smalltalk IDE, the language was shipped from the first day on together with an IDE that supports immediate search. This had an interesting consequence: Best practices that are related to text-editors never got adopted by the Smalltalk community, instead unique best practices emerged that are fully IDE-driven.

For example, it is common practice to mark pending tasks by calling a method named `#todo` (which is defined on Object and thus valid in all source code) rather than putting the term `todo` in a comment. In order to view all pending issues, developers just search for all references to the `Object#todo` method—and since the result open immediately, the developer will refer to this as "browsing all todos."

Using `todo` methods is not the only occurrence of such designated methods, the class `Object` is literally cluttered with designated methods. Even modifiers, such as *abstract*, are realized using a dedicated method call. The most remarkable of these designated methods is probably the `#halt` method that sets a breakpoint. As a developer, to set a breakpoint you would add a method call, to browse all breakpoints you would search for all references to `Object#halt`, and to disable a breakpoint you would comment out the corresponding method call.

So we observe that typically unrelated concepts such as tasks and breakpoints are unified through the socio-technical congruence of immediate search. A direct technical benefit of this unification is that design and implementation of the IDE become much simpler. An indirect benefit is that developers can create new tasks and breakpoints in the same way as they add methods to the system, there is no need to learn the arcane skills of how to write an IDE or VM plug-in.

For example, a conditional breakpoint that triggers only when shift is pressed can be implemented as follows

```
Object >> haltOnShift
    InputSensor shiftPressed ifTrue: [self halt]
```

One more example of designated methods: to deprecate a method, the developer adds a call to `#deprecated` to the method body. This is typically done as the first statement, such that the implementation of `#deprecated` may take appropriate action. For example, in a deployment context it might do nothing and in a development context it might pop up a warning dialog—or, even more interesting, it might automatically refactor the caller of the deprecated method to not use the deprecated method any more, which is also known as "inline renaming" [4].

4. CONCLUDING REMARKS

In this essay we outlined search-driven development and argued that human factors are as important as technical issue, but even more influence each other in unexpected ways. As an example, we looked at Smalltalk-80 (a system with immediate search-driven navigation) and observed the following socio-technical congruences

- We observed that in Smalltalk-80 searching for references and declarations is so fast and immediate that the user experience becomes one of browsing code rather than one of waiting for and clicking through search results
- We observed that, as a socio-technical consequence of immediate search, otherwise unrelated concepts (such as tasks and breakpoints) are unified into the same abstraction, *i.e.* method calls in the source code. Which, as another consequence, led to programming tools that are, at the same time, simpler and more extensible.

As a sidenote, we also observed how the co-evolution of programming language and development tools lead to immediate search results of Smalltalk-80's browse senders resp. implementers feature. And we also observed how the unilateral evolution of development tools (*i.e.* the development of a refactoring browser without according co-evolution of the language) led to a none-optimal search that might hinder the emergence of new socio-technical congruence.

With the rise of Eclipse, "compile" and "build" ceased to part of the user experience in Java development. Developers do not compile anymore, they immediately execute code. This feature was brought to Java from Smalltalk through the VisualAge heritage of Eclipse. As a personal conjecture, I'd say that our job as builders of IDE search tools is not done unless "search" ceases to be part of the user experience in software development (in the same way as "compile" ceased to be part of the software developer's user experience).

Acknowledgments.

We thank Niko Schwarz for his feedback and comments, and we thank the members of the Pharo mailing list for their help with Smalltalk and its search features. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010).

5. REFERENCES

[1] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[2] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. M. Drucker, and G. G. Robertson. Code thumbnails: Using spatial memory to navigate source code. In *VL/HCC*, pages 11–18, 2006.

[3] M. Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[4] T. Freese. Inline method considered helpful: An approach to interface evolution. page 1012. 2003.

[5] A. Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, 1984.

[6] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, May 1983.

[7] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[8] M. A. Hearst. *Search User Interfaces*. Cambridge University Press, 1 edition, September 2009.

[9] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, New York, NY, USA, 2005. ACM.

[10] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of ICSE'05*, pages 1–10, 2005.

[11] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52, 2008.

[12] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[13] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the 2004 conf. on Human factors in computing systems*, pages 151–158. ACM, 2004.

[14] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes. Codegenie: using test-cases to search and reuse source code. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 525–526, New York, NY, USA, 2007. ACM.

[15] A. Lienhard, J. Fierz, and O. Nierstrasz. Flow-centric, back-in-time debugging. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009*, volume 33 of *LNBIP*, pages 272–288. Springer, 2009.

[16] S. P. Reiss. Semantics-based code search. *Software Engineering, International Conference on*, 243–253, 2009.

[17] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[18] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 21. IBM Press, 1997.

[19] J. Starke, C. Luce, and J. Sillito. Working with search results. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 53–56, 2009.

[20] R. E. G. Valencia and S. E. Sim. Internet-scale code search. In *Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009. SUITE '09. ICSE Workshop on*, pages 49–52, 2009.

[21] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 513–523, USA, 2002. ACM.