More Archetypal Usage Scenarios for Software Search Engines

Werner Janjic, Oliver Hummel and Colin Atkinson

Software Engineering Group
University of Mannheim
68131 Mannheim, Germany

{werner|hummel|atkinson}@informatik.uni-mannheim.de

ABSTRACT

The increasing availability of software in all kinds of repositories has renewed interest in software retrieval and software reuse. Not only has there been significant progress in developing various types of tools for searching for reusable artifacts, but also the integration of these tools into development environments has matured considerably. Yet, relatively little is known on why and how developers use these features and whether there are applications of the technology that go beyond classic reuse. Since we believe it is important for our fledgling community to understand how developers can benefit from software search systems, we present an initial collection of archetypal usage scenarios for them. These are derived from a survey of existing literature along with novel ideas from ongoing experiments with a state of the art software search engine.

Categories and Subject Descriptors

D.2.13 [**Software**]: Reusable Software—*Reuse Models*

General Terms

Software Search Engines, Life Cycle, Reuse, Testing

Keywords

Software, Search Engines, Test-Driven Reuse, Discrepancy-Driven Testing

1. INTRODUCTION

As software becomes a ubiquitous element of our environment and daily lives, the quantity of available source code and components continues to grow steadily. Supported by the "open-source revolution" millions of software artifacts have become publicly available. This recently triggered the development of numerous internet-scale code and component search engines (see e.g. [6] for an overview). However, most of these search engines follow the "Google paradigm" and merely offer a simple keyword-based search interface of the form found in general web search engines. Although these efforts are clearly a step in the right direction, in order to better leverage the steadily increasing amount of software, it is obvious that software search engines need to offer more than plain text search. During the 1980s and 1990s, basic research on software retrieval made use of the specific features of software (such as operation signatures) and the survey of Mili et al. [9] provides a comprehensive overview of the state of the field about twelve years ago. Since then the ever growing amount of software artifacts available in company repositories and over the internet has triggered the development of more sophisticated retrieval approaches such as Component Rank [7] or test-driven reuse [3].

In spite of the progress mentioned above, little effort has been invested in investigating potential usage scenarios for software search engines that go beyond mere reuse in the coding phase of software development. In addition to our own fledgling thoughts in this area [3] we are only aware of two publications by Sim et al. and Umarji et al. [10, 11] that describe surveys amongst developers and maintainers published in 1998 respectively 2008. Although they provide valuable insights on the current usage of source code search engines, they did not elaborate on the further potential of software search engines in general. Nevertheless, our own preliminary experience suggests that software search engines have a tremendous potential to improve almost all phases of software development (i.e. analysis, design, coding, testing, deployment, and maintenance). Thus, we believe it is very important for our community to identify further usage scenarios that pave the way for innovative applications of software search and retrieval solutions in the future. Consequently, not only our community can benefit from a detailed investigation of the motivation, goals and problems in these usage scenarios, but eventually the productivity of software development and maintenance in general may increase.

In the remainder of this paper, we give an overview of currently recognized archetypal usage scenarios for source code search engines in section 2, describe innovative usage scenarios we identified from ongoing research, from experience in the development of the internet-scale software search engine Merobase and summarize them in section 3. We then conclude our contribution in section 4.

2. RELATED WORK

To date, research on, and development of, software search engines has been mainly driven by the goal formulated by Douglas McIlroy [8] – namely, to avoid re-inventing the wheel over and over again but to rather reuse existing artifacts. This general impression is clearly backed-up by the recent online survey conducted by Umarji et al. [11] who identified nine archetypal groups of source code searches amongst 58 anecdotes of how developers typically use source code search engines. Eight of these nine archetypes deal with reuse and the authors were able to distinguish four subgroups of searches for reusable code from four subgroups with searches for reference examples that are supposed to deliver some inspiration for how to implement some given programming task. Both groups can be further divided by the size of the artifacts ranging from small code snippets over object-sized units and libraries to complete standalone systems. One prominent exception is the inspiration that can be drawn from libraries, as developers are typically more interested in how to use existing libraries than in creating new ones. Consequently, this is an area that has attracted more research effort such as e.g. by Holmes and Murphy [2]. The ninth motivation in the named survey for why developers use source code search engines is to find a solution or a workaround for known defects e.g. patches in open source systems. However, experience tells us that general web search engines, indexing mailing list archives and forums, might be better suited for this job.

Further archetypal usage scenarios for source code search are known from an older survey by Sim et al. targeting on searches within a developer's immediate environment [10]. This also identified impact analysis for intended changes and program understanding as important motivations in this context. Although these two archetypes have not made their way into the more recent survey from above (perhaps because the required information is typically not available on an inter-project level), we nevertheless believe they will gain importance even for internet-scale repositories in the future.

3. MORE USAGE SCENARIOS

In this section we go beyond the aforementioned archetypal usage scenarios and identify additional ones. These enhance the existing scenarios or complement them with new and innovative ideas. The main driver behind our considerations was the motivation to better justify the large effort required for the development and set-up of software search solutions by extending the scope of their application. As indicated in the introduction, it seems likely that software search engines will not only be able to support coding, but other software development phases as well. In our opinion, distinguishing only reuse and inspiration as the main drivers for the use of software searching in a strict binary fashion does not fully justice this view. Thus we propose instead to imagine this as a spectrum ranging from rather speculative searches towards fully specified definitive searches such as described by Hummel [3]. Our understanding is that from a reuse point of view searches are rather speculative early in the software development process (i.e. late in the analysis or early in the design phase), since developers are then using software search engines to get an idea of what potentially reusable material is available or to get some inspiration on how to solve a given task. Later in the process (i.e. late in the design phase or during coding) searches can become much more definitive as typically a concrete specification of a required component is available. And in the sense of the KobrA component model [1] (which uses components on all levels of granularity and follows the premise that "one man's system can be another man's component") we prefer to avoid a distinction between the sizes of reusable artifacts.

Thus, we subdiveded this section into two reuse related subsections and a third subsection that contains the usage scenarios that are "maintenance-driven". We finally round off our observations in Subsection 3.4 with a visual summary of all identified usage scenarios assigned to those phases in the software development process where they are particularly useful.

3.1 Speculative Searches

The obvious and certainly still dominant usage scenarios for software search engines are those which come under the notion of *speculative searches*. In this context the search engine is used to get an impression of what is available in a repository and to provide a first idea of what might be a good design for a component the developer intends to write. At this point of time neither a detailed syntactical description nor a clear description of the semantics is likely to be available for the component under discussion. The developer makes use of the search engine to support the design process of the component being developed and to get support in the early implementation phase. Speculative searches subsume methods like the *design prompter* (see below), or drawing inspiration from open source and library searches as described in [11]. Searches for reusable code snippets also identified in the cited source are the only small exception in this context as they typically only appear while developers are already coding. All these searches are typically keyword-based searches, making little or no use of additional characteristics of code, such as interface descriptions or functionality specifications, and tend to occur during the design phase of a software project.

As mentioned in [5] the keyword-based searches typically used at this stage tend to be rather imprecise, but since the developer himself has no clear picture of the component which has to be written, this is not a crucial problem but rather a logical consequence.

3.1.1 Design Prompter

Given the progress that was achieved in data mining and related areas in recent years, it certainly makes sense to contemplate the automated creation of design hints based on existing software collections. Like the shop systems of large online retailers a proactive *design prompter* system might for example suggest to a developer that "other developers that have created a stack component also assigned a push and a pop method to it" [3]. Such a system needs to monitor the developer while he is designing or coding a system and then can make its recommendations based on the "mean value" of artifacts that other developers created in similar situations. This idea is not necessarily limited to the class level, it also seems feasible to extract helpful design or even architectural patterns from the contents of a software repository.

3.2 Definitive Searches

As soon as a software systems's design has become concrete enough that the "contours" of its components are clear, the requirements for a search engine change significantly as it can now utilize the additional features that the system and component specifications provide. In the following subsections we discuss the potential usage scenarios that arise from this group of definitive searches.

3.2.1 *Interface-based Searches*

Experiments published previously [5] have shown that ordinary keyword-based (even pure signature-based) searches (see Mili et al. [9] for detailed explanations) in internet-scale repositories do not deliver reusable material with a satisfactory level of precision. The main reason for this is the large number of candidates that can be returned for a request with a general keyword or signature. The quest for a more effective method consequently leads to the combination of signature and keyword matching, yielding the full consideration of all interface elements (i.e. typically its operation signatures) for a component search. In other words, interface-based searches utilize the syntactic features of software components for a search. They can be applied with operation (or service) signatures on the lower end of the size spectrum as well as with objects or full-grown components at the upper end. Although the experiments cited above reveal that the precision of these interface-based searches is already better, it is still rather imprecise as it offers no means to verify the semantics of components. This makes it practically impossible to discover a reuse candidate with different identifiers in its interface even though it might offer the required functionality.

3.2.2 *Test-Driven Reuse*

The identified lack of precision in simpler retrieval techniques was the main driver for the development of test-driven reuse [3] which exploited the fact that code documents differ from ordinary text documents by being executable and thus behaviourally observable. Though this approach is not as perfect as a specification in a formal language, test cases have been identified as a means to describe functionality with a common development artifact that needs to be created in the course of a development process anyway. In the case of so-called test-driven development, often used in agile development approaches, test cases are even used to drive component design and are thus created in an early development phase. The usage of such test cases in test-driven reuse effectively adds another filtering step on top of an underlying retrieval technique which can be chosen according to the size of the used repository, for example, increasing the precision of the whole retrieval process. If a large collection of components is available – and thus the probability for finding a matching reuse candidate is high – it makes sense to employ rather strict pre-filtering (such as an interface-based search), while a small collection on the other hand might benefit more from pre-filtering with signature matching in order to also test those candidates that might be adaptable by more sophisticated tools [3].

3.2.3 *Discrepancy-Driven Testing*

Another approach for utilizing software search engines based on a definitive description of the expected results can be applied during software testing. Often such things as licensing issues or company regulations (or simply the infamous ”not invented here syndrome”) prohibit the integration of reusable components into a system. However, it would be a pity not to utilize the knowledge bound up in such components. An elegant approach to still take advantage of discovered software is to use retrieved components (e.g. by a test-driven reuse tool) as test oracles. As soon as components that are functionally equivalent to the one under development have been discovered, they can be executed alongside the component under test with a large amount of randomly created input values and the delivered output values of all components can be compared with each other [4]. As soon as a disagreement occurs between them it is clear that an interesting test case has been discovered that is worth being investigated manually as obviously at least one developer has already made a mistake in this context before. This testing approach is (with a set of manually created components) known as back-to-back testing [12]. The aim of *discrepancy-driven testing* is to avoid the expensive manual creation of test cases and thus to allow discovering more defects in a shorter period of time as is possible with traditional testing approaches.

3.2.4 *Test Case Discovery*

As software search engines clearly not only contain components, but often also the test cases intended to test them (e.g. about 100.000 JUnit test cases are reported by the Merobase search engine), expanding on the idea of discrepancy-driven testing one step ahead is certainly interesting. In other words, even if we do not want to apply the approach to a component, we can still extract the knowledge that is stored within the numerous test cases available for similar components. If we assume that test cases contain the knowledge of domain experts (which it is very likely to happen in company repositories), this usage scenario is not only appealing in order to achieve better software testing with less effort, it might also be interesting to use the data contained in the test cases in order to improve software specifications. The combination of this approach with discrepancy-driven testing is still immature, but nevertheless seems promising and thus is under active research within our group.

3.2.5 *Library Searches*

During the development of software systems, developers often need to incorporate additional external libraries into their projects. Software search engines can be used in a variety of ways to increase the productivity of developers in this context. First, when the documentation is not sufficient to efficiently use a library, tools such as Strathcona [2] can be used to discover code examples of how other developers have used the given library. Second, a developer may be forced to look for the source code of a designated class from a specific open source system to better comprehend the way it works (or perhaps even to verify a bug). A search engine which is able to deliver the required source quickly is likely to save a lot of effort that otherwise would have to be invested into locating and downloading the appropriate file in some open source hosting system on the Web. A third way of using search in the context of software libraries is the possibility to use it for finding a library containing a specific class. Such a search often becomes necessary e.g. due to Java's well-known `ClassNotFoundException`s, produced when the classpath is not set correctly during compilation or deployment. Due to the numerous frameworks and components available today, identifying, finding and retrieving the correct library can become a tedious process that is likely to become more efficient with the dedicated support of a search engine.

3.3 Maintenance Driven Uses

The year 2000 problem demonstrated impressively that software systems operate for a long time and thus mainte-

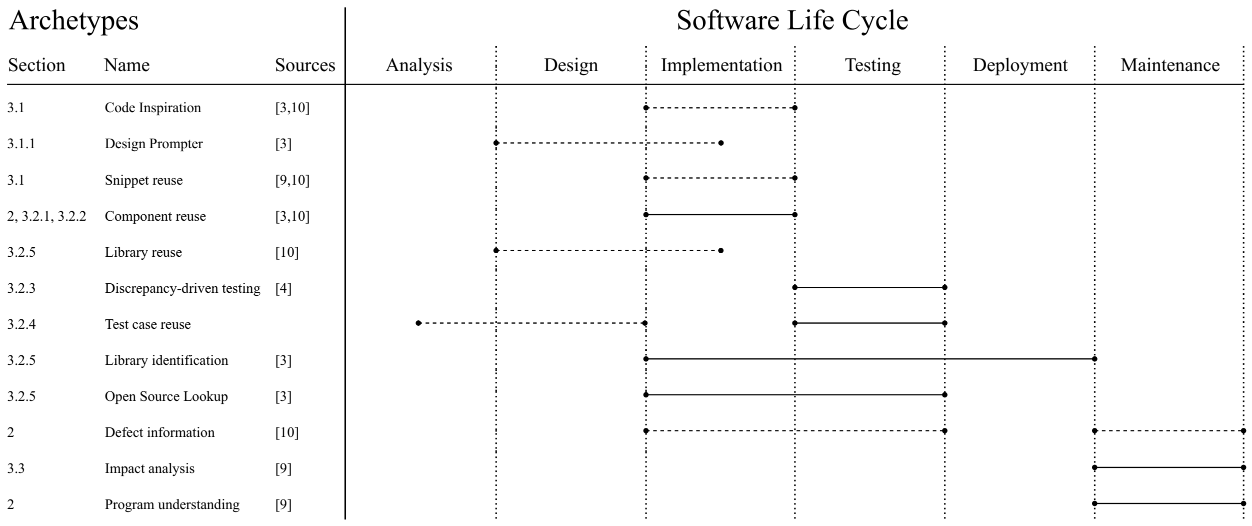

Figure 1: Usage scenarios for software search engines against the software development lifecycle

nance is a crucial activity in all software systems' life cycles. Consequently, developers identified already in the late 1990s that analyzing the impacts of a change to a system is an important application that needs to be supported by software search [10]. Today this can be done with common IDEs such as Eclipse, but it also seems feasible to extend the range of such a functionality to an internet-scale code base.

3.4 Summary

In this paper we have added a number of new usage scenarios beyond the classic archetypes known from the literature, most of which are currently under intensive research. Therefore it is still hard to give concrete recommendations about when the use of these approaches makes the most sense within the software development process. Nevertheless, we try to summarize our findings in Figure 1 which contains an overview of the usage archetypes we have identified in the first column on the left-hand side. The first row on top of the figure lists the software development phases and the lines in the central area of the figure indicate when a usage scenario is potentially useful in a given phase. Furthermore, the figure distinguishes between speculative searches (dashed lines) and definitive searches (solid lines).

4. CONCLUSION

In this paper we have identified a lack of a systematic investigation of potential usage scenarios for software search engines. Although the literature provides two surveys where developers have been asked how they are currently utilizing these search engines, we believe the recent advances in software retrieval technology opens much more potential for further interesting applications of this technology, which "normal" developers are not yet able to imagine.

Thus, after briefly surveying the existing literature on usage scenarios for software search we complemented the resulting list with further scenarios we discovered during our (partially ongoing) research on software retrieval. The resulting collection is intended as a starting point to develop new and innovative applications for software search engines and thus to better justify the large costs involved in setting-up and operating such systems.

5. REFERENCES

[1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-Based Product Line Engineering with UML*. Addison Wesley, 2001.

[2] R. Holmes, R. J. Walker, and G. C. Murphy. Strathcona example recommendation tool. In *Proc. of the 10th Europ. SE conf.*, 2005.

[3] O. Hummel. *Semantic Component Retrieval in Software Engineering*. PhD thesis, University of Mannheim, 2008.

[4] O. Hummel, C. Atkinson, D. Brenner, and S. Keklik. Improving Testing Efficiency through Component Harvesting. In *Proc. Brazilian Workshop on Component Based Development*, 2006.

[5] O. Hummel, W. Janjic, and C. Atkinson. Evaluating the efficiency of retrieval methods for component repositories. In *Proc. of the 19th Intl. Conf. on Software Engineering & Knowledge Engineering.*

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

[7] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3), 2005.

[8] M. McIlroy. Mass Produced Software Components. Software Engineering. In *Report on a conf. sponsored by the NATO Science Committee*, 1968.

[9] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of SE*, 5, 1998.

[10] S. Sim, C. Clarke, and R. Holt. Archetypal source code searches: A survey of software developers and maintainers. *Intl Conf. on Prog. Compr.*, 1998.

[11] M. Umarji, S. Sim, and C. Lopes. Archetypal internet-scale source code searching. *Proc. of IFIP World Comp. Congr. on Open Source Software*, 2008.

[12] M. Vouk. On back-to-back testing. *Intl. Conf. on Computer Assurance*, 1988.