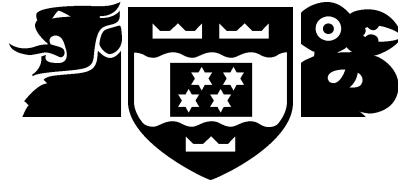


VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

An Architecture for Behaviour-based Component Retrieval

Ashley Schroder

Supervisor: Stuart Marshall

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

The user of a large repository of reusable components faces two major difficulties while discovering suitable components. Firstly they must accurately describe the desired behaviour, and secondly they must find that behaviour within the repository. Behaviour-based searching is one technique that helps overcome these difficulties. In this report an improved behaviour-based search technique is developed that addresses several key limitations of earlier work, in particular focusing on applying the technique to object-oriented components. This technique is incorporated into a search architecture for tool support that further addresses several limitations with behaviour searching feasibility. To determine whether the new architecture, and our improved technique are feasible, a prototype of the architecture was developed, called codeCollective. This prototype was functional enough to allow early testing to be performed, which shows promising results in several tests of speed and accuracy. This report will present the motivation for, and result of, an improved search technique for object-oriented components, demonstrating that the prototype is both responsive and accurate.

Acknowledgments

Thanks to Stuart Marshall for all his help and support year, his advice and commitment have been invaluable to me. I couldn't have done it without him.

Thanks to Cindy whose company through the long, long honours nights was often all that kept from throwing in the towel. Thanks to Bridie for her regular call-ins to offer support, energy drinks or to tell drunken tales. Thanks to Mum and Dad for putting up with my visits to Palmy, just to eat their food and watch their TV.

Thanks to Chris, Christo, Darren, Hayden, Tats and the rest of the inmates of Mephis lab for the great times.

Contents

1	Introduction	3
1.1	Project Overview	3
1.2	Background	4
1.3	Report Organization	5
2	Motivation	7
2.1	Text-Based Searches	7
2.2	Formal Searches	8
2.3	Behaviour-Based Searches	9
2.3.1	Behaviour sampling	11
2.3.2	Generalized Behaviour-based Retrieval(GBR)	11
2.3.3	Modern Behaviour-based Searching	12
2.3.4	Behaviour Searching and Testing	12
2.4	Limitations of Behavioural Searching	13
2.5	Contributions	13
3	An Improved Behaviour Search	15
3.1	Overview of the Search Technique	15
3.2	Rationalizing the Improved Search	18
3.2.1	Object-oriented Limitations	18
3.2.2	Applicability Limitations	21
3.3	Un-addressed Limits and Issues	22
4	The Architecture	23
4.1	Overview	23
4.2	Peer	24
4.3	Index	24
4.4	Search Engine	25
4.5	Network	25
4.6	User Interface	26
4.7	Architecture Feasibility	27
5	Implementing a Prototype	29
5.1	How the prototype was developed	29
5.2	The Implemented Modules	30
5.2.1	Index	30
5.2.2	Search Engine	31
5.2.3	User interface	32
5.2.4	Network	34
5.3	Prototype Limitations	34

5.4	Prototype Summary	35
6	Testing	37
6.1	Test Setup	37
6.2	Indexing Speed	37
6.2.1	Method	38
6.2.2	Hypothesis	38
6.2.3	Results	38
6.2.4	Interpretation	38
6.3	Searching Speed	38
6.3.1	Method	39
6.3.2	Hypothesis	39
6.3.3	Results	39
6.4	Search Accuracy	40
6.4.1	Method	40
6.4.2	Hypothesis	40
6.4.3	Results	40
7	Discussion	43
7.1	Findings	43
7.1.1	Index Speed Findings	43
7.1.2	Search Speed Findings	44
7.1.3	Search Execution Findings	45
7.2	The Wider Implications	46
7.2.1	Benefits	46
7.2.2	Disadvantages	47
7.2.3	Applicability to the SDLC	47
8	Summary	49
8.1	Future Work	49
8.1.1	Areas of Improvement	49
8.1.2	Areas of application	50
8.2	Contributions	50
8.3	Conclusion	50

Figures

1.1	The behaviour searching process	5
2.1	The trade-off between ease of use and accuracy	10
3.1	The Search Process outline	15
3.2	The Reuse Process	16
3.3	The Chaining Algorithm	17
3.4	The Invoke Algorithm	18
4.1	The Structure of the Architecture	25
4.2	The User Interface	26
5.1	The Index	30
5.2	The Search Interface	33
6.1	Graph comparing number of classes with time	39

List of Tables

2.1	Informal text search quality	8
2.2	Equivalent behavioural search queries	9
6.1	Indexing Test Results	38
6.2	Searching Test Results	40
6.3	Searching Test Results	41

Chapter 1

Introduction

Reusing existing software can have several benefits for software development [18, 11]. When we reuse software we avoid having to re-design, re-implement and re-test it. Unfortunately though, widespread reuse is not common within the software engineering industry, so these potential benefits are not realized [13]. This is because there are two key difficulties with reusing software: component discovery and component integration. Component discovery is describing, finding and evaluating a component needed to perform the required task. Component integration is process of altering the component found to operate with other components. This project will focus on the difficulties with component discovery, and how they can be dealt with by partly automating the component search and retrieval process. In particular, we propose and investigate a behaviour-based search technique that exploits the executability of a component.

1.1 Project Overview

We are interested in searching for reusable components. For the purposes of our project, we use Vitharanas' definition of a component: "a chunk of code that offers services to users" [21]. We see the component as a black box, that provides external services. The services of a component are the individual units of external behaviour that are offered. A component may be accompanied by source code and documentation and this helps to reuse it. However, in many cases these additional sources of information are not present.

An important distinction to draw is the difference between components written for reuse and components being reused where they were not originally intended for reuse. Our proposed search technique supports searching for both types of components. Testing in chapter six will show that components intended for reuse take longer to search, but obviously provide flexible functionality. This project will address the limitations with behaviour-based searches by making three key contributions:

- The development of a behaviour-based search technique for modern object-oriented components. This will address the limited output description capability of behaviour searching, allowing the range of output represented, and subsequently searchable, to grow to include output from complex object-oriented components.
- This new technique will be incorporated into the design of a tool support architecture to address issues with usability, scalability and computational cost associated with conducting a dynamic behaviour-based search.
- This architectural design will be prototyped for use with Java. This prototype will serve as a *proof of concept* which allows testing to be conducted.

- Testing will be performed with the prototype to investigate the feasibility of the search technique and the architecture.

The net result of these contributions will be a prototypical tool-support architecture for modern, dynamic behaviour-based searching of object-oriented components. The extent to which this architecture will benefit reuse will be tested and discussed, along with the feasibility of the search technique in general.

1.2 Background

Over the past few decades there have been numerous attempts at various approaches to simplify the task of component discovery [10]. Overcoming the difficulties associated with describing, storing and finding components will go a long way to easing the burden of reuse on developers.

The various approaches to this problem tend to exhibit an inversely proportional relationship between simplicity and accuracy. The higher the accuracy a user wants, the less simple the technique will be and vice-versa. Research in the area is largely divided into two main contingents.

The formal software development approach is to give precise specifications of components, and search a repository for matching specifications. This formalism provides unparalleled accuracy, but the overhead of creating these formal specifications makes it difficult to use such techniques in industrial situations.

An alternative technique is text-based searching. The user provides a natural language query, and a search of the repository returns potential matches. This is very easy to prepare and execute, but the accuracy suffers, because of the inherent ambiguity in natural language.

Behaviour-based searching is a technique that exploits the executability of components to determine their behaviour. The user provides one or more samples of desirable input-output mappings. These mappings are used during a dynamic search. The services offered by a component are executed with the given input, and the output returned is inspected for comparability to the desired output in the mapping. An example of this process is illustrated in Figure 1.1.

Behaviour-based searching represents a convenient middle-ground between these two approaches. The technique lacks the high degree of accuracy that comes with rigorous formalism, and the ease of use that comes with natural language querying. Behaviour-based searching presents the opportunity to gain an element of both ease of use and relatively high accuracy, a combination which neither formal specification searching nor text-based querying can offer.

The basic premise of the behaviour search is that by describing desired behaviour with example input-output mappings, we can determine the suitability of components in a repository through invocation. The search proceeds by supplying the given input to each potential candidate component, the component is invoked using the provided input, and the output value is compared to the user provided example output. If the two are *comparable* then there is a high chance that the given service implements the desired functionality. Unlike formal theorem prover based techniques, there is no verification that the component implements the required behaviour. The user must manually inspect the suggested components.

Like any technique for finding components, behaviour searching is not flawless, it has by nature several limitations which make it difficult to apply to a modern development process. These limitations are: a narrow scope for describing output, a lack of any evidence of feasibility in a modern environment, and a lack of consideration for security and scalability. We address these limitations so as to reduce the impact they have on the applicability of

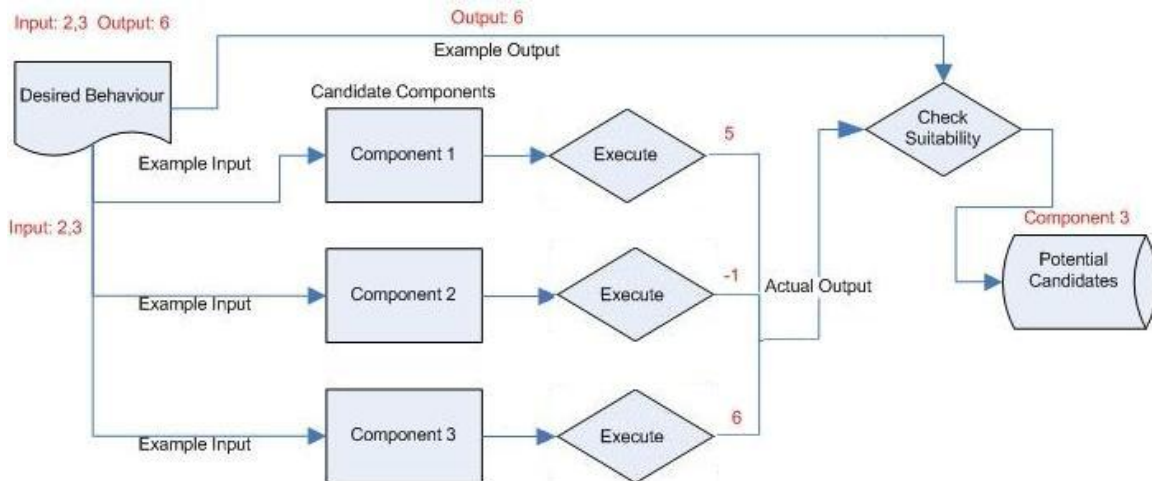


Figure 1.1: The behaviour searching process: In this example we are searching for a component service that multiplies two numbers. By executing the example input on each component, we observe the actual output and compare it to the desired output.

behaviour-based searches. With better techniques for discovering suitable components in a repository, we can alleviate the associated difficulty with reuse.

1.3 Report Organization

The remainder of this project report is organized as follows. Chapter two will discuss the related works in detail, motivating an improvement in current behaviour-based searching techniques. Chapter three will look at the development of the new behaviour-based search technique for object-oriented components. Chapter four will propose the design of an architecture for tools supporting the use of our new technique in a manner that addresses several limitations with behaviour-based searches. Chapters five and six will look at the implementation and testing of this architecture. Chapter seven will discuss the implications of the test results in detail, indicating the feasibility exhibited by the early prototype.

Chapter 2

Motivation

The goal of efficiently retrieving useful components from a repository has been pursued by many researchers utilizing a wide variety of methods [10]. There is currently no single search technique that will solve all re-use needs. This research will propose and investigate a new behaviour-based search technique that balances ease of use and accuracy to meet the needs of modern software development. This chapter will look at a broad overview of different search methods, in a bid to better understand how a behaviour-based search provides a careful balance between accuracy and ease of use. This survey will examine the strengths and weaknesses of text-based searches, looking at how the use of natural language is easy, but inaccurate. Formal searches will also be examined, these are costly in time and effort, but provide unparalleled accuracy. This will lead lastly to an introduction to behaviour-based searching. We will look at the history of behaviour-based search techniques, and the limitations with the current state of research in this area.

2.1 Text-Based Searches

Text searching currently forms the basis for the industry standard in component searching [20]. This is due in the most part to the ease of use afforded by natural language queries.

The brute force text-search approach has been improved through the use of indexing and classification [17, 6]. Indexing allows the pre-hashing of keywords that appear regularly in a source code or documentation resource. This dramatically boosts speed of lookup. Classification is another technique that improves searching by grouping words based on meaning. The fact remains though, that the use of a natural language to describe components is fraught with difficulties.

The accuracy of such queries is low due to the ambiguity of natural language. It relies on the code or documentation having keywords that describe the behaviour, and obviously that behaviour description is consistent between developers. A developer seeking a `sorting` algorithm, will probably miss the `ordering` algorithm in the repository, despite the two algorithms providing identical behaviour.

The problem with a text search, when being used to index software is that text fails to capture the behaviour of components, instead it can only classify the language used to subjectively describe them. This ambiguity in natural language descriptions leads to searches that will potentially have an inversely proportionate relationship between recall and precision [19]. Text-based searches cannot understand binary, so they rely on classifying components by their source code and documentation. As outlined earlier this limits the components that can be searched as there are situations where components are not well supported by source code or documentation.

An industry example of a keyword-based search is `www.koders.com`. This is a massive open-source code repository that searches for code containing given keywords. Figure 2.1 indicates the results of informal text searches using the Koders database of components. It is fast and easy to use, but a simple search for code that generates permutations highlights the difficulties with text searching.

A search for the keyword `permutations` generates 400 results. On the first page of these results, 21 of the 25 results appeared by manual inspection to be completely irrelevant. This inaccuracy and false-positive rate makes it difficult for users to determine suitability.

Search String	# of Results	First page false-positives	Estimated rate
<code>permutations</code>	394	21	0.84
<code>sorting</code>	6731	19	0.76
<code>file rename</code>	17,800	14	0.56
Average	8308	18	0.72

Table 2.1: Informal text search quality

This characterizes the ambiguity in natural language searches outlined earlier. The results can vary in accuracy due to the imprecise description used to perform the search. If a consistent vocabulary between developers of components could be established, this ambiguity would be less of an issue.

However, these difficulties are offset by the fact that text-based searches are both easy to implement and simple to use. This is highlighted by the popularity of text searching in all areas of information retrieval. The huge success of web-search engines such as Google, highlight the attractiveness of a simple text search.

2.2 Formal Searches

Formal methods researchers have also tackled the retrieval of components. Components are described with formal specifications [14, 23, 7, 9], the developer using the repository must provide a formal specification-based query. This generally allows a form of theorem prover to show that the component implements the desired requirements. This is a powerful concept, because the components determined to meet the specification verifiably implement the desired behaviour.

The benefits of a formal specification search lay in the potential for the accuracy to be perfect if the specification given correctly describes the desired behaviour. This accuracy is a major advantage and is important in development where verification is a key requirement.

However the problem is that the developer must be able to specify precisely the desired behaviour, and the components must have been submitted with the necessary information. This adds considerable overhead to a search, meaning that for most developers it is not worth spending time generating a specification to search with. This is also extra overhead for authors of components, who must spend time specifying the precise behaviour of each service, to facilitate formal searches.

Another shortcoming of formal searches that are built around specification provers is the time they take to execute a search, which can be considerable. In a recent survey of formal model checking technology results indicated lengthy execution times in the order of hours [4]. This model checking has similar computational complexity to component specification

matching, and is indicative of the high cost to perform these theorem proofs. This rules out searches for any sort of sizable component repository where the cost to attempt proofs of all possible components would be too high.

The formal search uses several interesting techniques. One such technique is a refinement ordering that allows a lattice of partially ordered components to be formed. This concept is a powerful way to facilitate retrieval as it provides scope for close matches and approximations. Likewise less rigorous specifications can aid the usability of such techniques, but the ease of use comes at the cost of accuracy as less formal specifications cannot provide the same well-defined mapping to implementations, and subsequently lose the verifiability provided by well-defined refinement approaches.

We can see that though formal searches offer unparalleled accuracy, this accuracy comes at the cost of speed and usability. In many ways this is the polar opposite to text searching, where searches have no verifiable accuracy but simple operation and fast execution.

2.3 Behaviour-Based Searches

The spectrum of retrieval approaches, starts at the imprecise but simple to use and implement text search, and ends at the provably accurate but cumbersome formal search. Somewhere between overt simplicity and absolute formalism lies behaviour-based searching.

Behaviour searching is motivated by the simple tenet that by searching for components using textual descriptions, we are failing to utilize the fundamental difference between plain text and code: the property of *executability*. To better show a components suitability for a certain required functionality we can match the desired behaviour with the actual behaviour. We will look at several of the early implementations of behavioural searches, identifying weaknesses and innovation.

Behaviour-based searches provide more accurate results than a text only search, and are less rigorous than a formal specification search. Behaviour-based searching is a convenient middle-ground that foregoes the accuracy of a formal search to leverage ease of use and simplicity, more in line with the text based search. The components gathered are not proven to be correct, so the user must still inspect the suggested candidates for suitability. The user must also construct a more complex query than a text search, by providing example input and output. This trade-off has been shown on a set of axes in Figure 2.1

To illustrate how a behaviour-based search would be constructed, we present example searches in Table 2.2. Consider the text searches in Table 2.1 and how these parallel the following behavioural queries.

Search String	Input	Output
permutations	Collection={1,2,3,4,5}	Collection={1,3,5,4,2}
sorting	Collection={5,2,3,1,4}	Collection={1,2,3,4,5}
file rename	file=/home/test.txt, string="test2.txt"	file=/home/test2.txt

Table 2.2: Equivalent behavioural searches to the text searches trialed in Table 2.1.

These searches will not uniquely define the exact service required. This is clear as there are many more permutation mappings than just the given one. There are many other possible sortings, or services that will output the given ordering. This ambiguity becomes less of an issue as we provide more examples. In the extreme case, if we provide *all* examples,

The Inherent Trade-off between Ease of Use and Accuracy

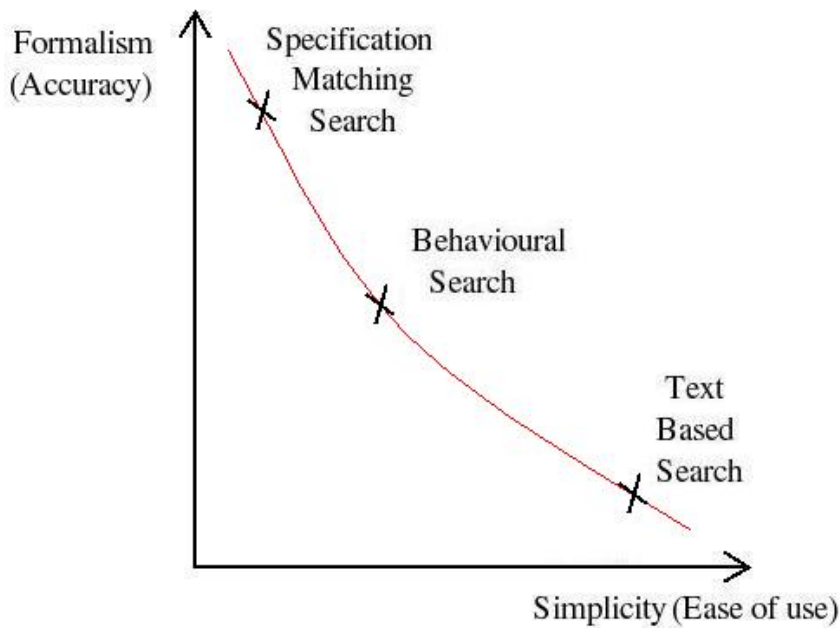


Figure 2.1: The trade-off between ease of use and accuracy. Notice that behaviour-based searches represent a balanced trade-off, being neither overly cumbersome nor exceedingly inaccurate

then our proof by example technique, is verifiable. The more example behaviour provided, the more accurate a search can be.

There is still a formal foundation for a behavioural retrieval. Atkinson and Duke use a lattice structure [1] similar to that proposed by Milli et al in [14]. Atkinson and Duke's formalism allows for the extension of behaviour, and provides a partial ordering that allows for close behavioural matches. Although a powerful formal definition of the behaviour search, it does not deal with output that comes from anywhere but a return value. This is one of the key limitations that an improved technique must address.

Behaviour based searching accuracy is limited by the fact that execution of anything less than an entire set of possible inputs will fail to demonstrate its behaviour in all cases. This is shown formally below.

For Components X and Y , equivalence classes of input A and B and example input a .

$$\begin{aligned} &\exists a \in A, \exists b \in B \\ &input(a, X) \rightarrow output(a, X) \\ &input(a, Y) \rightarrow output(a, Y) \end{aligned}$$

In this case the two components would both be considered suitable, as both perform the

required mapping. However, in certain cases this is demonstrably unsound. Consider:

$$\begin{aligned}
 &in(A, X) \cap in(B, Y) \neq \emptyset \\
 &\quad \textit{therefore} \\
 &\quad \exists a \in A \textit{ such that} \\
 &\quad in(a, X) \rightarrow out(a, X) \\
 &\quad in(a, Y) \rightarrow out(b, Y)
 \end{aligned}$$

Furthermore there can be errors in our determination of suitable components if we have the situation where one input set is a proper subset of another, consider:

$$\begin{aligned}
 &in(A, X) \subset in(A, Y) \\
 &\quad \textit{therefore} \\
 &\quad \exists a \in A \bullet \\
 &\quad in(a, X) \rightarrow out(a, X) \\
 &\quad in(a, Y) \rightarrow out(a, Y)
 \end{aligned}$$

These limitations are inherent in behaviour-based searching techniques, due to that fact that testing one example input represents a *proof by example* that a component meets a requirement. It is important not to consider a behavioural search as an entire evaluation of that component, it is an aid in component discovery. The purpose of this aid is to drastically reduce the search space for the developers manual inspection.

2.3.1 Behaviour sampling

Podgurski and Pierce originally developed *behavioural sampling* [16] which was an early attempt at exploiting executability to help describe components. The term sampling comes from the random sample of potential inputs it generates. The concept behind behaviour sampling was simple: the developer would provide the desired input type and output type, and a random sample of the input values would be generated. For each of these inputs, the developer would provide a desirable output. These randomly generated inputs would be provided to the services of a component, and the return value from each would be compared to the calculated output from the developer.

Behaviour sampling itself presented an innovative approach, but had several shortcomings that made it ultimately an infeasible search method. A general failure to allow for the complex nature of component side-effects, meant that candidate components were not being adequately described by their actual behaviour, but a smaller, less accurate subset. The input is selected from a random distribution of the domain, which means that it may bear no resemblance to potential special cases. The decision on input distribution should be made intelligently by the user. Behaviour sampling also only invoked one method to search for component functionality. This means that the functionality of a class that combines several services, is never found.

2.3.2 Generalized Behaviour-based Retrieval(GBR)

Generalized Behavior-based Retrieval(GBR) introduced by Hall[5] provided a considerable number of improvements to Behavior Sampling. His proposal was to model the components

in a repository as functions and simply execute those functions with the provided input. By representing the library as a collection of functions Hall managed to overcome many of the shortcomings in Behavior Sampling. GBR can simulate the environment (such as the filesystem) on which side-effects could happen, and observe changes. GBR also uses type information to reduce the search space by several orders of magnitude. Combining methods together to provide functionality that could not otherwise be found in behaviour sampling is a massive improvement in potential recall, at the expense of computational cost.

As useful as these improvements are to the behaviour-based searching in general, they come at the cost of a pre-engineered library that consists of functional representations of components, not classes and methods. There is no notion of the object-oriented concept of encapsulated state and behavior. For use in modern object-oriented repositories GBR would need to be significantly altered to allow object state to be represented along with the functional representation of behaviour. In addition, the sub-typing relationship present in object-oriented languages means that the simple process of signature matching will not be as easily able to rule out a candidate component.

2.3.3 Modern Behaviour-based Searching

There have been few published attempts to extend behaviour-based searches to allow for object oriented component searching. More recent attempts at behavior searches on Object-oriented libraries [15] seem to take a simplistic view of the behaviour-based search, ignoring many of the ideas Hall proposed in GBR and certainly simplifying the complex output capabilities of object-oriented components. One of the issues that Niu and Park deal with is the sub-classing and sub-typing relationships that classes in a repository will often be part of. This allows for substitutable types in arguments and return values to be accommodated. They also mention tolerable return values, indicating some sort of close approximation has been implemented, this would be related to Atkinson and Dukes' formalized behavioural approximation, though this is not detailed in Niu and Parks paper.

Squeak [8] is a Smalltalk [3] implementation that is written entirely in Smalltalk. Although not a widely used language, it provides a feature that closely mirrors a localized version of behaviour searching. Squeak has what is called a *methodFinder*, which can be used to search for local methods within the development environment that implement the desired functionality. Though this is a great feature, it has little application to more widely used languages, where the distinction between development environment, and operating system is more clear. Smalltalk is pure object-oriented language with powerful meta language support. this makes the implementation of such searches simpler as the queries are *part of the language*. The capabilities of *methodFinder* are similarly hobbled by a lack of concern for complex aliasing mutations and object state changes that provide avenues for component output (This is more or less an API search, but could be applied to a repository).

This survey of search techniques has highlighted the usefulness of a search that offers a balance of both usability and accuracy. The limitations of a behaviour-based search in terms of modern development with objects have also been identified. In Section 2.4 we will look at these limitations more closely.

2.3.4 Behaviour Searching and Testing

There are several similarities and differences between unit testing and behaviour searching. We will look briefly at these to see how the two concepts are closely related. A unit test is essentially a behavioural specification that is hard coded for one component. This test is then applied to the component that has implemented the necessary functionality, and if the

test is passed the component is considered to have successfully implemented the required behaviour. By analogy a behavioural search is a generic unit test, that can be executed on any component whose interface will permit it. Each component is executed and passing the test indicates the correct behaviour. If unit tests could be written more generically, then their application could be more widespread, given that the method names and objects would all be dynamically bound.

JUnit for example is a test framework for Java(www.junit.org). It represents a reverse behaviour search in a way. Components are written, and one or more test cases are paired with each. The tests run and define correct operation for each component. Any component that does not match its desired behaviour is considered to have failed its tests. The interesting feature that JUnit provides is a set-up and tear-down phase for each execution. This allows the test environment to create runtime objects to be used during testing. This will be discussed as a feature in more detail in Chapter 3.

2.4 Limitations of Behavioural Searching

Behaviour Searching is a promising approach for finding potentially reusable components without high developer overhead. There are several limitations with the current state of behaviour sampling that must be addressed for behaviour searching to be considered applicable to modern components. These are:

1. *Capturing output of an object-oriented component*
Methods can output in a number of possible ways, yet current behaviour-based searches only inspect return values. Output can come from object state changes, parameter mutations, external side effects and even exceptions. To increase the accuracy of a behaviour-based search, we must be able to capture these modes of output.
2. *Security*
Running unknown code in any situation is dangerous, and so techniques to alleviate this potential hazard must be sought. This is due simply to the fact that the code is unknown to us. It may not be malicious, or coded with malicious intent, but there are certainly potentially hazardous effects that come from executing programs which are untrusted, and for which we may not have source code.
3. *Scalability*
This technique is easy to imagine applied to five or ten components. The reality is that with today's software engineering requirements, the search must be able to scan many thousands of components. This requires the search to be fast when applied to large repositories, so that users need not wait lengthy periods for results.
4. *Empirical evidence supporting feasibility of behavioural searching*
In a modern object-oriented context, the question of whether such searches could be applied to large repositories is still open. An early indication would allow research to steer away if results were not promising.

2.5 Contributions

This project makes four key contributions to the field of behaviour-based searching:

1. Develop a new search technique that accommodates the varied output in modern components.

2. Incorporate this search in a language independent architecture that provides tool-support for fast, effective and safe searches to be conducted across distributed repositories.
3. A prototype tool will be implemented to demonstrate the search technique.
4. The prototype will be tested showing the feasibility of the object-oriented behaviour searching.

In the next chapter we will address limitation one by developing a new search technique that handles object-oriented components. In chapter four we design an architecture for tool-support of this search engine, which will address limitations two and three. And in chapters five and six we will build and test a prototype which will address limitation 4.

Chapter 3

An Improved Behaviour Search

This project will address four limitations identified in chapter two with behaviour-based searching approaches. Limitation one was to do with the handling the results of object-oriented component behaviour. This is addressed through the development of a new behaviour-based search, that allows developers to determine the suitability of an object-oriented component based on examples of the required behaviour. In this chapter we present our technique. The technique will be rationalized by identifying and addressing the limitations with previous behaviour-based searches.

3.1 Overview of the Search Technique

The search technique consists of four steps. These steps are shown in Figure 3.1. The technique does not cover the entire reuse process, as this involves storing and integrating components. The reuse process also often involves tools for aiding user inspection. Figure 3.2 shows our search technique in the wider scope of the reuse activity. In this section each of these four steps will be looked at in detail, so that the reader can understand our technique, and how it allows component services to be behaviourally inspected. To help the reader understand the steps we illustrated each step with an example.

In this chapter the word *candidate* will be used to describe a chain of one or more services offered by a component. This represents a possible chain that would map the users example input onto required output.

Name	Overview	Input	Output
Filter	Remove unsuitable Candidates	Candidate Set	Subset
Chain	Determine chains that provide the mapping	Subset	Chains
Invoke	Execute each chain and store its output	Chains	Actual Output
Check	Inspect the set of outputs for matches	Output	Set of matches

Figure 3.1: The Search Process outline

Our example will portray a developer who requires a component that provides statistical analysis functionality. The developer has a collection of integers for which the mean, median and range is sought. This is simple example, but allows us to demonstrate all four steps of our technique.

Behavioural Specification
 Mean: input={1,1,3,5,5,6,7}, output = 4
 Median: input={1,1,3,5,5,6,7}, output = 5
 Range: input={1,1,3,5,5,6,7}, output = 6
 Maximum depth: 2

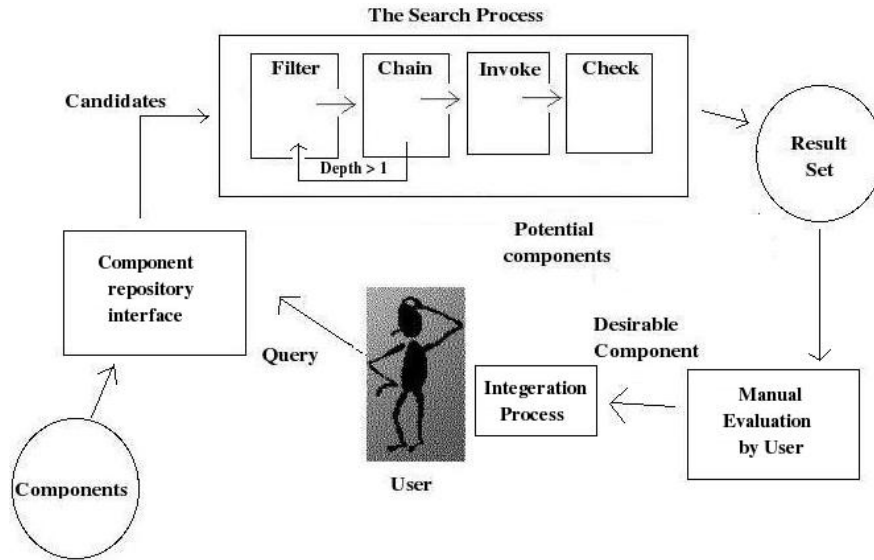


Figure 3.2: The reuse process as a whole. The scope of the new search is shown, ranging from obtaining a candidate from the repository, which is then filtered using the static index. The candidates are then selected for chaining and invoked. The results are inspected and potential matches are added to the result set.

The search would consist of one or more Input → Output mapping for each service required. These mappings specify behaviour desired from a service. In our example the developer will provide one example mapping for each required service. The developer also provides a maximum depth setting. The maximum depth setting controls how long the chains of service will be. This gives the user an element of control over the trade-off between accuracy and speed. A long chain length will result in a larger, more accurate search space but a longer search time. Each of the four steps are explained below:

1. Filter

The Index provides Candidates to the search engine. These Candidates ensure that the index information conforms to an interface, but also make the passing of this information from one place to another easier. The Candidate contains all of the statically available information, which is used to reduce the dynamic overhead. As each Candidate enters the process, it is examined to determine whether it could potentially provide the desired Input→Output mapping. The subset of all Candidates that can provide the necessary mapping, is output to the chaining step.

In our example, this means looking for all services which can take collections as parameters. This step does not look for output of single integers, because depth two has been selected. This means that in the next step, we will look for another sub-set of candidate chains that provide as output a single integer in one or two invocations.

```

In : desired input, desired output
begin chain
  if ! depth >1
    filter for candidates whos input/output matches desired
  else
    //user has specified to look for chains of functionality
    filter for candidates whose input is compatible with the current input
    set current candidates output to be new current input
    reduce depth by 1
    recursively call chain with new input and desired output
  end
end

```

Figure 3.3: The Chaining Algorithm shows the pseudo code overview of how the chaining is performed

2. Chain

In this step the maximum depth the user has specified becomes important. If the user has not specified a higher depth than one, the candidates filtered in the first step are further filtered to just those whose output is of the valid type. These are output to the invocation step. If the user has selected a level of chained method calls ≥ 1 , then this step will recursively search to the specified depth before returning. At each level of recursion, more methods will be added to the current chains. For each candidate that has the necessary input, but not the necessary output, a candidate is sought that can take the current output as input, and return the desired output.

In our example, let us assume we have found a `BoxAndWhisker` component. This component takes a collection of integers, and writes statistics about them to local fields. These initial methods may not return the values, so a depth one search would not uncover the potential functionality. If the component provided methods called `getMedian` and `getRange` that read the given fields, then these will be chained after the initial method that took the collection as input. This is because the output of the first method is considered to be the writing of a local field. This output is then considered input into the accessor methods that return the fields as output directly.

The algorithm for this step has been shown in pseudo code in Figure 3.3. The result of this chaining algorithm, is a set of method calls from components that will, by some pre-determined chain of execution, provide the required Input→Output mapping. These chains are passed to the invocation engine for execution.

3. Invoke

The dynamic invocation takes place in this step of the search process. Each chain has its example input objects instantiated and passed to each service, the result of which is collected, and either passed to the next service, if chained, or stored for comparison.

One of the important features in this part of the process is the permuting of substitutable inputs. For example if a method takes two Strings, the method will be invoked twice, once for each ordering the example strings. This decouples the implementation specifics of the candidate component from the behaviour specification.

For our example we would set up the collection object, containing the given integers. Then we would pass this to the invocation method, where it proceeds to change the underlying `BoxAndWhisker` object. This change of object state is considered output

```

In : desired input, desired output, Chain serviceList
begin Invoke
  for each chain
    set up input objects
    for each method in the current chain
      invoke with a permutation of current input
      store output object as current input

pass the chains and outputs to the check step

```

Figure 3.4: The Invoke Algorithm shows the pseudo-code overview of how the chaining is performed

from the first invocation, and a second method which uses this intermediate output as input to provide the required output, is invoked. This final output from the chain is stored and passed to the comparison step.

This algorithm is difficult to describe in natural language, it is presented in pseudo code in Figure 3.4.

4. Check

Each chain whose output matches the desired output is considered to provide the required functionality. These chains are added to the result set. This set is returned to the developer for manual evaluation.

In our example, we would find the chains who returned the field values that the `BoxAndWhisker` object produced when a collection was passed to one of its methods.

The definition of 'matching output' could potentially be extended so that partial matches are included and ranked.

The example chosen was purposefully illustrative of the difficulties with natural language descriptions. If we had searched for 'Statistics', or 'Mean' in a text-based search, the `BoxAndWhisker` class would have been low on the list as the name does not contain either keyword. However the behaviour represents exactly the functionality required. This exemplifies why behaviour-based searching can be beneficial.

3.2 Rationalizing the Improved Search

The limitations of behaviour-based searching techniques have been briefly introduced in chapter two. In this chapter we address the technical issues that cause these limitations.

There are two types of limit, firstly there are limitations related to the fact that behaviour sampling has no history of widespread application to object-oriented components. Secondly there are limitations related to weaknesses in the current technique that affect the applicability in *real-world* development.

3.2.1 Object-oriented Limitations

Object-oriented limitations are due in the most part to the complexities of how object-oriented component behaviour causes output and side-effects. This was not dealt with well by previous research into behaviour-based searching.

Object Output

Objects present a rich variety of output capabilities, and are not limited to just return values. Object-oriented components can output in one of three ways:

1. External output- such as to file system, network or display
2. Object state/program output - such as parameter mutation or field update
3. Exceptional output - such as exception throwing

In this search engine we have primarily dealt with the second type of output. We discuss dealing with output one and three in the next section, but in general the search algorithm described here only considers object state changes and return values. External state changes need to be accommodated outside of the bounds of simple search engine, by providing a layer of abstraction between the search engine and the physical system. Extending the search engine tool-support to accommodate these outputs will be discussed in chapter four.

1. *Representing desired behaviour*

The user is required to specify the way in which they wish to observe behaviour. If they are seeking simple input-output transforming components then they need not inspect the changes to object state or method parameters. If they seek more complicated components that alter the object state or mutate the parameters, then the user needs to specify given initial and desired final values for these objects. This seems to unnecessarily complicate the query forming process but it allows a more complex range of components to be characterized. This complex behaviour was lacking in earlier behaviour-based searches, which failed to allow for the encapsulation of object state and behaviour.

2. *Observing Output*

Once selected for execution a candidate's behaviour will be observed in any of three key areas. The exact way in which a component will be inspected depends on the structure of the user query. The list below presents the three potential changes that can be observed.

- **Return Values**

The first, and most obvious is the output of a method that returns a value. This can be directly compared to the desired output provided by the searcher and potential matches can be reported, for some given definition of matching.

- **Object State**

The second area in which we observe changes, is to the runtime state of the underlying object. We can determine the changes that occur inside the object upon which the method was invoked. This allows us to select candidate classes that display some desirable behaviour upon invocation of certain methods. This is performed by setting the internal fields of the object to known values before invocation, and inspecting them afterwards.

- **Parameter Mutation**

This type of dynamic checking seeks to find a method that performs some operation on an object that is passed to it. We know the state of the input parameters as they are provided by the user, we then inspect how they are changed after invocation.

External Side Effects and Exceptions

Object services may have external side-effects. A service that outwardly appears to only perform simple processing operations, may in fact be making changes to filesystem, network or display state as well.

Components that exhibit side effects are difficult to handle in behavior based searches. These external changes are things such as network or filesystem communication, or user interface prompts.

GBR approaches the problem by providing virtual filesystems with known initial state. However GBR does not present a solution to the problem of acceptable failure cases and user input/output. In the new behaviour search, we have allowed for an abstract layer that can detect certain external side-effects as output. Certain external effects such as GUI prompts will still cause problems. A solution is to simply suppress this type of output.

Certain situations may also cause program execution to throw exceptions. These are generally considered erroneous, but in object oriented programming they can be a type of message passing. Developers should be able to specify the desirable behaviour of exception throwing. This can be handled by instrumenting the catching of exceptions and determining if the thrown exceptions were desired.

Combining Services

The single service of a component may be incapable of providing the required behaviour. When combined with one or more other services, the component may produce desirable results.

One of the improvements made by GBR is the potential for methods to be chained together to form a combined behaviour. While the desired behaviour may not be provided by any single method, there may exist a sequence of methods that satisfy the desired behavior sample. Where each method utilizes output from previous methods as input. This concept is similar to the more recent concept of Jungloids[12]. Jungloids are paths from one type to another through multiple methods. They however do not use dynamic behaviour to disambiguate the multiple paths through a type graph, relying instead on heuristics.

As presented in the analysis of GBR, the complexity of this chained invocation can be high. There is potential for many millions of possible combinations to exist from a set of input to output behavior. This is offset by restricting the methods that will be invoked based on their signatures, and by not distinguishing between components that provide the same output.

In this search technique, we must distinguish between a policy of chaining services based on class, or based on component. For example if we do not limit chains to only services offered by the same class, then we allow the potential for chains such as the following from a `String` to a `URLConnection` Object:

```
URL yahoo = new URL("http://www.yahoo.com/");
URLConnection yahooConnection = yahoo.openConnection();
```

If we chain method calls only from within a class, we reduce the complexity of the chaining procedure drastically. We could also limit chains of service from different classes within the same component to provide the desired functionality. The choice here is between speed and accuracy.

Forming queries

The way in which the developer describes queries directly affects the range of components they can search. If we use an integrated approach, where the querying is performed in the same language and environment in which the programs will run, we can construct any object and use it as example input and output. This kind of integration is seen in languages such as Squeak. This allows the user to specify any sort of behaviour in that given language. The challenge though is being able to make that integration work. We can model our set-up and tear-down procedures in a similar way to JUnit where before and after each invocation the objects are created and destroyed. This will add extra cost to the querying process, but will allow larger scope for object description.

3.2.2 Applicability Limitations

Applicability limitations are due to issues with behaviour sampling that make it difficult to apply to situations in the real world, or for more complicated components.

Search Speed and Cost

This affects the developers perception of usability. If searches take too long to perform, then they will not be considered viable. We must ensure that our searches are comparably responsive to other developer tasks such as compilation. The solution to this need for fast searching is to index the necessary information statically. This will be detailed in the next two chapters.

Non-termination

If the input to a service causes it to not terminate, or worse still, just to run for a very long time the search responsiveness will be compromised. It has been proven that we cannot *solve* non-termination. We can reduce the effect it has on the overall search process though.

The real problem is that we have to choose a time at which we decide to stop waiting and move on to the next service. This really needs some developer input, as they will be the best judge of how long they should wait for a certain algorithm. If they are looking for an algorithm to calculate a very high prime number, they should expect to wait several years.

GBR proposes two potential solutions, the use of preconditions that describe acceptable input values or a time-out length. Both increase the time it takes to perform a search.

The decidability of termination is an open problem in mathematics, logic and computer science. The best our behaviour-based search can do is avoid running indefinitely by terminating after some user-defined acceptable time.

Non-determinism

Non-determinism is a problem because the result of a method invocation with the same input may give two different outputs. Consider two program statements, S and T. If S is executed in parallel to T then the result is equivalent to non-deterministically choosing the execution order. If this is not dealt with using concurrency primitives, then the changing order of instructions may have an impact on the output.

$$S \parallel T \equiv S; T \square T; S$$

This is problematic in terms of re-use because components may appear to meet the requirements during one execution, but after that may not provide the desired functionality.

It could be an issue in any component where multiple threads are running. We could detect multi-threading during execution, and notify the developer who is searching, that the component may be non-deterministic.

3.3 Un-addressed Limits and Issues

The technique presented is by no means the last word in behaviour-based searching. It presents a way for object-oriented components to be behaviourally searched and that addresses limitations identified in chapter two. There are several limitations that still persist with the given technique. These are related to chaining complexity, speed, invocation with exceptions, permutation complexity and security. The next chapter will look to reduce the impact of these limitations by addressing them in a tool-support architecture for this new search technique.

Chapter 4

The Architecture

In the previous chapter we introduced a search technique for use in an object-oriented component repository. This search technique has a wider scope for object output. Behaviour searching has limitations that are not addressed by the new search technique. In this chapter we will present an architecture for tool-support of our new search technique.

There are three main limitations this tool will address: speed, security and scalability. These relate roughly to limitations two and three identified in chapter two. The architecture will be designed as language independent, allowing implementation in any language where technology supports behavioural searching.

4.1 Overview

The tool-support architecture consists of four key parts:

1. The static index
2. The search engine
3. The user interface
4. The network

This chapter will look at each of these four modules, outlining their role in the system and their internal operation. The benefits and limitations of each module will be discussed, giving an insight into the rationale behind the design decisions made.

The architecture's design supports the improved search presented in chapter three. In addition it provides several key features that increase the feasibility of behaviour-based searches. The architecture offers the following benefits:

- *A reduction in the search cost*
The goal is to make the response time for a single search, from user query to result printing, as low as possible. This is achieved through the use of a statically available repository index and the distribution of searches across a peer-to-peer network of repositories.
- *A improvement to scalability*
The search time per class needs to be the same when searching five classes as it is when searching five thousand. This scalability will ensure that users who are searching large repositories can still access that information in a reasonable time. This scalability is facilitated by the use of the network, which distributes execution cost across multiple

users. The static index allows the search space to reduce dramatically without expensive dynamic checking.

- *Reducing the security threat*

The risk with foreign code is that, whether maliciously or accidentally, it can cause damage to the system executing it. This can be offset in a number of ways dependent on the chosen platform for implementation.

At the architectural level of the tool, a way of minimizing this risk is to force the execution to occur away from the developer searching for the component. This is achieved through the distribution of the repositories. The authors sharing their own components are responsible for ensuring they do not pose a threat. This is because it is their own system on which these components are executed.

- *User interaction with the Search engine*

The search engine itself has a complicated interface. Having a user interface that can map human readable behaviour descriptions into the search queries used by the search engine, can make the process faster for developers. This interface could be provided in a number of ways, ranging from the integrated approach of Squeak, or a command-line approach such as in GBR. The main focus is on providing the mapping from human behavioural description to search engine description.

Figure 4.1 presents a graphical overview of the system. The modules in this figure will be looked at in more detail in the following sections.

4.2 Peer

The peer in the architectural design represents a single user or dedicated repository. This user can initiate searches that are sent to both the local search engine, and to remote search engines within the peer-to-peer network. The extent to which a peer is a part of the network is entirely decided by that user. If a developer only wants to search their personal code repository (or third party libraries) then they need not connect to the network.

4.3 Index

The index is where pre-fetched component information is stored and accessed. The index quickly provides candidates to the search engine, that have been filtered, based on the statically accessible information stored. This means that the dynamic cost of a search is reduced.

The index is an important module in terms speed and scalability. By providing near constant access time to a large body of component information, the per component search cost is considerably lower. This relates directly to the speed of a search as this high information extraction cost must be paid at some stage. By paying for it *in advance* we effectively shift the burden away from the search process.

The implication of this design is that there is a high cost to add components to a repository, because they must be inspected at the time of inclusion. The rational behind this decision stems from the desire to have high responsiveness in the search. The indexing process occurs offline, as a background process. Searching is an interactive process, requiring the fastest possible response time.

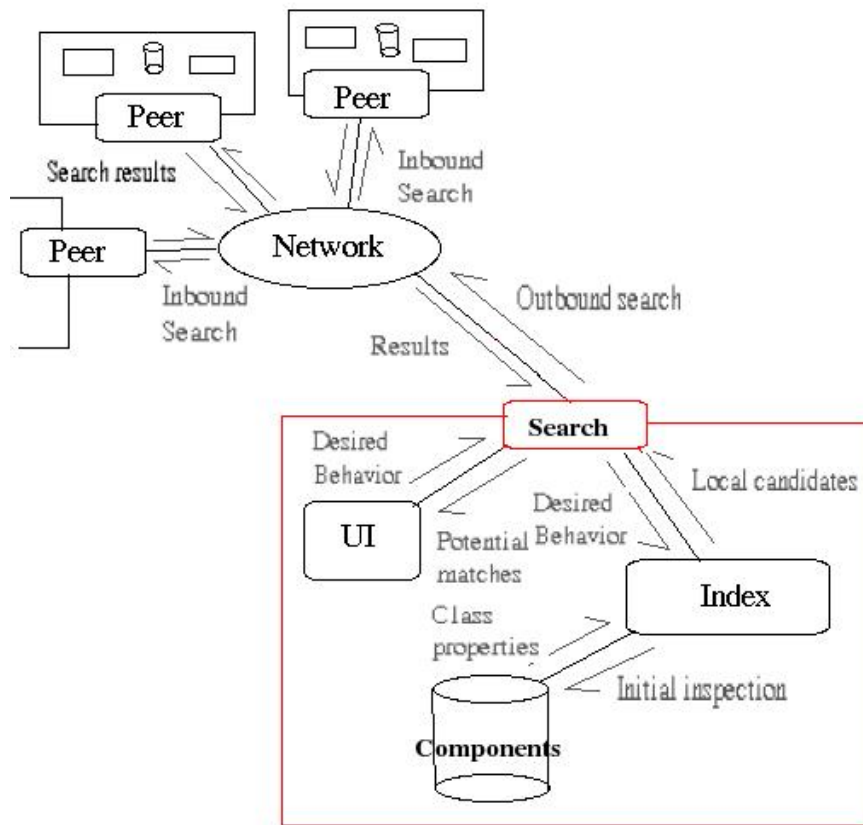


Figure 4.1: The Structure of the Architecture showing how the search engine, user interface, index and network interact to provide the reuse architecture that utilizes behaviour searches.

4.4 Search Engine

This module represents the search engine described in Chapter 3. It is the central hub of activity within the tool. The required behaviour arrives from the user interface, in a prepared format. This is combined with the filtered candidates from the index. The candidate is invoked with the given input and the output is checked. This outputs a result set, containing any candidates whose behaviour matched the required behaviour specification.

It is important that the well defined interfaces are maintained. This allows the various modules that interact with the search engine to evolve, while the engine itself continues to function as it should. Similarly this modularity also means that the search engine can alter its process, while still maintaining the outward interface. In doing so it allows easy upgrade or adaption, without requiring alteration to any other parts of the system.

4.5 Network

The network represents the connection of one peer to another. The distribution of this tool over a peer-to-peer network provides several key benefits.

- The speed of a search is increased because the cost of execution is distributed across the peers where the candidate components are located.

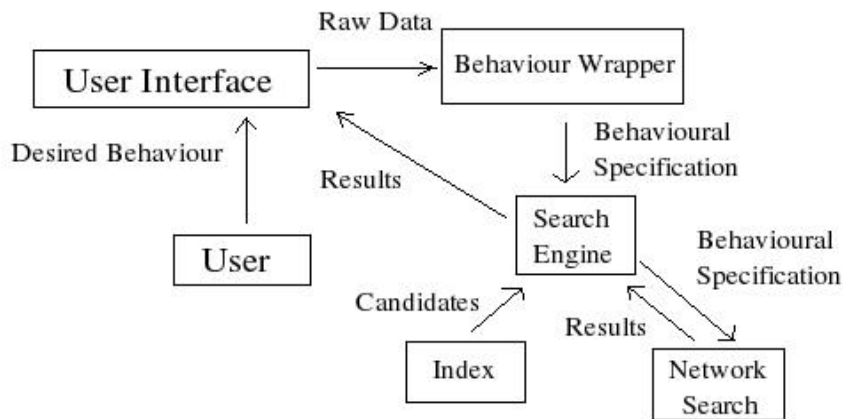


Figure 4.2: The flow of information from the user to the search engine. The GUI is responsible for wrapping the raw required behaviour into the requirements specification that the search engine will use.

- The security increases because the foreign code is not executed on the local system. This places the burden of security with the user who shares dangerous code.
- The scalability of the search is guaranteed by virtue of the fact that each additional code repository comes with computational power to run searches.
- Of less importance is the the hand selection of code to share on the network. Each user may improve the general *re-usability* of components in the system by having chosen them for sharing manually.

The network module provides the community aspect of the architecture. The extent to which a user wanted to participate would differ depending on their goals. Users who wish to simply use the behavioural search functionality to inspect a local library or API, would require no network functionality. On the other hand there is potential for the network to represent a community of developers who share amongst themselves useful domain specific utility components.

The network could be implemented in a number of ways, the actual technology is unimportant. It is simply a medium for the behaviour specification to be delivered to remote search engines. The network acts as a gateway passing information to and from remote search engines, in much the same was as the user interface passes that information to and from local users.

4.6 User Interface

The user interface takes the developer's query and maps it onto an object that is understood by the search engine. The user interface information flow is illustrated in Figure 4.2. It begins with the developer specifying a query, which is processed and sent in a structured form to the search engine. This modularity means the user interface can be altered or adapted. Provided the interface remains unchanged the new user interface will not make any difference to the search engine.

The nature of the user interface would depend largely on the implementation technology. In Squeak (discussed in chapter two) we identified the unique inclusion of the querying

language in the development environment. GBR uses a separate command-line user interface. Both of these would be adequate for this architecture. We feel the queries written in the language to which they apply are the most expressive. This allows the actual objects involved to be instantiated in the same runtime environment although it does make the queries language specific. This trade-off would need to be analysed in terms of usability.

4.7 Architecture Feasibility

The architecture presented in this chapter is designed to achieve fast, secure and scalable behaviour searching in a modern development environment by addressing several key limitations with existing behaviour-based searches. This is achieved by having each module in the tool designed to benefit the behavioural search process. The focus in the next two chapters now turns to feasibility. This design will be implemented and we will test the implementation for feasibility.

Chapter 5

Implementing codeCollective: A prototype for tool-support

In the previous chapter we outlined the design for a tool support architecture, that facilitates the behavioural searching of object-oriented component repositories. In this chapter we will prototype an implementation of this architecture called *codeCollective* which serves as a proof of concept for both the search technique, and the tool-support.

The purpose of this implementation is to allow a demonstration of feasibility, through basic testing. This in turn will provide empirical evidence for the feasibility of our behavioural search technique. Recall a lack of evidence regarding the feasibility of object-oriented behaviour-based searches was identified as a limitation of the technique.

In this chapter we begin by giving a broad overview of the implementation process, how it proceeded and what the end result was. This will lead to a closer examination of each module of the design, describing the technology used to implement it. The limitations and benefits of the chosen technology used for each module will be discussed. This chapter reports on the successes and failures of the implementation process, with respect to the goal of providing a test-ready prototype with which to demonstrate feasibility.

5.1 How the prototype was developed

The prototype was implemented using Java, chosen for its provision of a meta-level programming API, which facilitates the behavioural searching we require. This prototype applies to searching Java classes because the reflection API is limited to Java classes.

The prototype was developed iteratively, starting with a crude prototypical spike that served as a chance to probe for new technology or new capabilities within old technology. This first prototype was experimental, but it paved the way for a completely new implementation which was built closely following the design. The objects communicate through a series of well-defined interfaces, this meant the new architecture was modular and easily extensible.

The requirements for the development were prioritized. Implementing the new object-oriented search technique and the index module were primary goals. A usable interface and peer-to-peer networking had to be secondary goals as time would not permit a fully functioning prototype of the architecture to be developed. Though incomplete, the priority placed on the two most important modules meant the prototype allowed several key tests to be carried out. These tests illustrate the merit in continuing development of the architecture.

In the next section we will discuss how the architecture design was mapped to an implementation. We will detail the technology used, and how that choice of technology proved

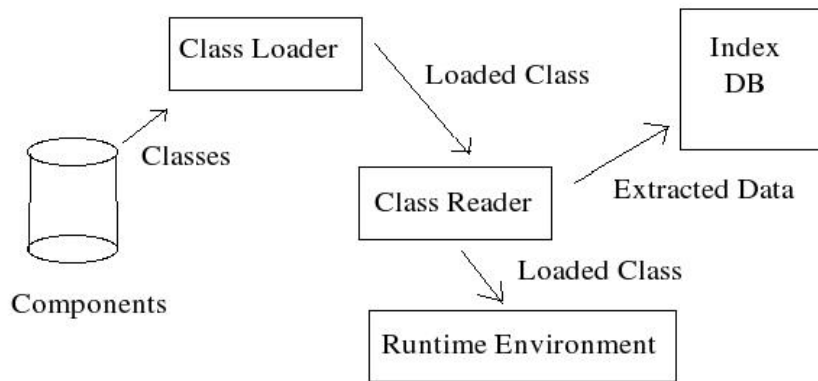


Figure 5.1: The Index reads in the raw classes, loads them and extracts information about each one to store in the statically accessed Index database.

to be beneficial or limiting. This will lead to discussing how the limitations of the prototype affect the validity of testing.

5.2 The Implemented Modules

Each module had a clear role to play in the system, dictated by the architectural design. In this section we will explain how each module in the design was implemented. We conclude by identifying the limitations of each module and discussing ways in which these limitations can be addressed in future development iterations.

5.2.1 Index

The index is an important part of the architecture as it provides both speed and scalability by reducing the runtime burden on the search engine. This is because the heavy cost of determining component characteristics is performed in advance of the actual search. This increases both the speed of searching and the scalability of searches on larger repositories. The basic operation of the index module is shown in Figure 5.1

- **Technology:**
The index was implemented in one of the early prototypical spikes as a flat-file database, where each line is a space delimited collection of candidate information. This proved to be a fast, effective solution in the early stages of development. Now the simplicity of the index is a hindrance due to it's inability to evolve.
- **Benefits:**
The index is very fast, because it is memory resident for most of the applications life-time. This would not be the case in systems where memory is shared with many other processes or memory is in short supply. Small flat-file databases represents very raw input/output speed.

This simple database was also very easy to implement in the early stages, this meant that a working prototype index was put together quickly and accommodated the necessary functions of the early search engine.

- **Disadvantages:**
With the increased need for information indexing, the flat-file database proved to be problematic. Any changes to the structure of the database were coupled to changes in the code that accesses the database. This meant that adding statically determined chaining relationships would be very difficult.
- **Limitations:**
The index in its current form is not able to handle the pre-calculating of potential chains. This currently has to be done dynamically. This has a high cost associated with it as the calculating of valid chains can often be quite complex. Because of the primitive nature of the database, changes to the structure of index require altering code that accesses the information. This is a hurdle to any improvements to the stored content.
- **Improvements:**
This index could be improved by migrating to a relational database such as PostgreSQL. This would accommodate a more complex query language, which allows selection of candidates based on chain forming ability and input/output type in linear order time. This also de-couples the storing of information from the retrieving of it, which could then be done using the JDBC driver system.

5.2.2 Search Engine

The search engine module is where the heavy dynamic behavioural searching occurs. Its operation has been defined in chapter three, and the design was detailed in chapter four.

- **Technology:**
The search takes advantage of the meta-level programming API provided by the `java.lang.reflection` library. This has been a feature of Java since version 1.1. Meta-level programming is a feature that is lacking in many other modern object-oriented languages, which meant that the choice to use Java was mainly a result of this provision.
- **Benefits:**
Java reflection is a powerful library of code that lets the programmer access the structures that the language is built from: methods, fields and classes. This means that we can instantiate program constructs based on dynamically retrieved information and have the results obtained and compared.
- **Disadvantages:**
The documentation and online help available for reflection is not extensive, and it became apparent during implementation that the reflection library has not been used in a manner that we were using it for by very many people. This made debugging difficult, because there was a lack of support.

Reflection has a few cumbersome traits, it allows invoking methods with either the primitive, or its respective object wrapper. It will not allow finding the methods in a class based on the same approximation (called auto-boxing). This means that at various stages in the search engine, the example data and indeed the type of that data must be manipulated for use with reflection.
- **Limitations:**
The search engine has several key limitations. It failed to properly implement the

chaining code, as a result of the complex recursion having subtle bugs. This is not an issue for depth less than or equal to one.

The actual concrete implementation of chaining in the search engine is based only on seeking chains within classes. As outlined in chapter three, there is more accuracy to be gained by including chains from within the component. This additional cost would have to be migrated to the indexing stage, if the search was to remain feasible.

The search engine also hasn't dealt with the invoked methods exception throwing very well. Technically the throwing of an exception in object-oriented languages such as Java can be considered a form of output. The developer should be able to specify conditions in which a `NullPointerException`, or `ArrayOutOfBoundsException` should occur, and use these as a way of partitioning the set of input into good and bad equivalence classes.

Another issue is with invoked methods of a class making method calls, this is possible if the input needed for this sub-routine is available, but will cause problems if it is not in the virtual machine at invocation time. There is not really an easy way around this, because using reflection we cannot inspect the actual commands of a class. This could be dealt with using static analysis techniques, searching the call graph statically and identifying components that make sub-calls. The required input to those callees could be 'set-up' when the actual service is invoked.

- **Improvements:**
The search engine is very much prototypical, it was sufficient for feasibility testing. For industrial application it would need several more development iterations. These could address the issues with the chaining and the potential for exceptions to be considered as output. Both issues affect the robustness of the current prototype, but I feel it has served as a reasonable indication of the feasibility in its current form.

5.2.3 User interface

The user interface is responsible for mapping human readable queries into the behavioural specification understood by the search engine. In this prototype the user interface was a simple SWT front end, that took information from text fields and used it to build the specification object. The prototype interface with search output is shown in Figure 5.2.

- **Technology:**
The technology used was SWT user interface API, and to a lesser extent the layout and structure was built with the help of SWT designer. SWT is a flexible, fine-grained GUI API, the full extent of its potential was never realized by the simple user interface implemented.
- **Benefits:**
The SWT designer offers fast development, in a similar way to Visual Basic or Visual C++. This meant that time could be spent on the mapping accuracy rather than how the user interface should be constructed.
- **Disadvantages:**
The separation of query from the virtual machine means that a whole subset of potential queries is made difficult. This relates back to the way in which we describe required specification, discussed in chapter 3 and chapter 4.

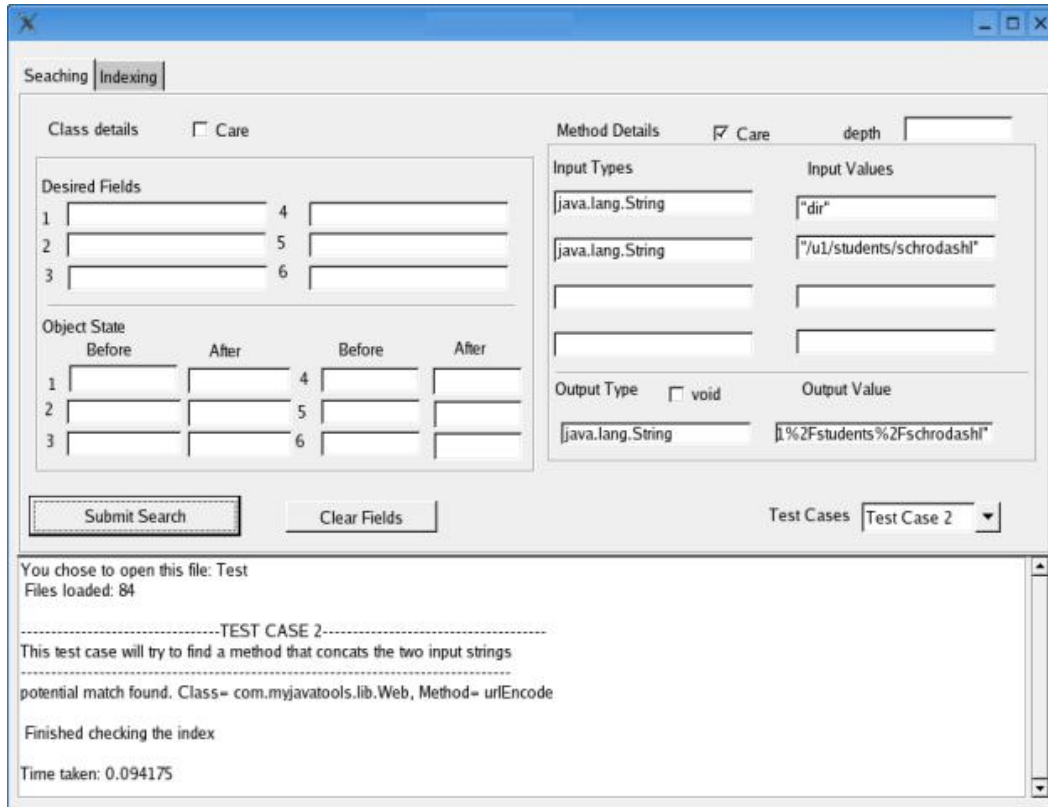


Figure 5.2: The Search Interface shows how the user inputs type and state information for a query. The left pane represents the object state before and after invocation, the right pane represents the parameters type and values.

Basically by having natural text input, we make it easier for the user, but the actual objects involved must be instantiated based on the text, which means objects that cannot be instantiated (or reliably instantiated) using text, are difficult to describe.

- Limitations:
The primary limitation has been outlined above. There is no textual description that can map onto complex non-literal objects such as abstract ideas like syntax trees. Generally speaking, if we can construct an object from a string, it is literal, and can be described in this system. Most objects can be described with strings, but it becomes difficult for objects that do not lend themselves to string representation (such as images).
- Improvements:
Adopting a Squeak style query interface that is built into the runtime environment, would allow objects involved in a query to be instantiated from file or from the network with as much ease as they can be if they were being instantiated for use in an actual program.

One possible way of easily including this set-up and tear-down style querying would be to use AspectJ to separate this concern. We could have before advice on the invoke method, that constructs some pre-defined objects and places references to them in the required array of input. After invocation we could simply delete all of the temporary

objects. This would be a simple solution to the limitation on Java's runtime environment, which would allow a rich range of components to be specified.

5.2.4 Network

The network forms the link between each search engine and accompanying repository. This element of the architecture is important in terms of applicability, however because of the focus on developing a prototype that demonstrates feasibility, the network was not implemented.

- **Technology:**
Though not implemented, research was performed on how a network module could be implemented. The Java JXTA framework for p2p applications was identified as a potential solution. Another alternative would be to utilize an existing large scale p2p network and simply operate within that interface.

- **Benefits:**
The way the search engine has been set up to receive a well formed behaviour specification, means that the network can act as a pseudo interface to many such engines across remote repositories. This is hidden from the local modules of the architecture, and the network simply acts like the model as in.

For example:

```
view.display(model.search(specification) +  
network.search(specification))
```

would be a simple way of outputting both local and network search results. The network module would co-ordinate the sending and receiving of the search and its results.

- **Disadvantages:**
The network would need to be secure and responsive for the search time to be considered feasible when searches are sent across the network. Security is an issue, because malicious users may be able to send the incorrect code back to a user who selects what they believe to be a suitable component. This could have extremely dangerous results. There needs to be a well defined protocol for digitally signing (or verifying with hash values) the content of a component.
- **Improvements:**
Implementing a prototype of the network would allow the scope of feasibility testing to be extended.

5.3 Prototype Limitations

The prototype itself still has several major limitations. The chaining algorithm is incorrect, often reporting impossible chains or not recursing appropriately. This needs to be addressed with further development and testing. This chaining fault is an issue in terms of feasibility for the prototype, but also in terms of the test quality. Because it is unreliable it could potentially skew the results.

The search itself can fail in certain cases, mainly due to Java reflection having peculiar behaviour surrounding sub-method invocation and exception handling within code that is reflectively invoked. This is an issue in terms of feasibility and test data quality also.

The network and user interface both need additional work as indicated in the previous section. However these are not major factors in the feasibility testing. The user interface

relates more to usability and the network is simply an extension of the index and search engine to other users. Feasibility testing for one search engine and repository should scale to a network of such search engines.

5.4 Prototype Summary

In this chapter we have presented the implementation of codeCollective, a prototype of tool-support of our new behavioural search. We have discussed the technology utilized during development and the benefits and disadvantages of that technology. We looked at the limitations with the current implementation and the implications these limits will have on our feasibility testing in the chapter six.

Chapter 6

Testing

In this chapter we are testing the prototype, the implementation of which was presented in chapter five. This prototype represents an incomplete implementation of the architecture presented in chapter four. The purpose of this testing is to address limitation four identified in chapter two, the current lack of empirical evidence regarding the feasibility of behaviour-based searches that deal with object-oriented components.

These tests of feasibility will focus on speed and accuracy. Though not a complete definition of feasible, we have selected the speed and accuracy of the search as two key factors which decide whether it is of use to users. Speed is the measure of both the time it takes to actually have the query respond, but also of the time it takes to index a repository. These two activities are by far the most computationally expensive, and so if we can demonstrate that these times are within acceptable bounds, then we would have shown that the search is potentially feasible.

Testing will not be concerned with the usability of the prototype. This cannot be adequately tested given the current state of the prototype. Usability is certainly a concern for the feasibility of the tool, however it would require engineering of a usable interface. This was not a priority during prototyping, and so testing it here would be a fruitless task.

6.1 Test Setup

The tests will be conducted on a simple NetBSD workstation, running a Pentium 4 2.8GHz processor with 256MB of RAM. The timing tests will make use of the system hardware timer. The test repositories will be a range of sizes gathered from the class corpus. We accept that these large jar files are not the best representative of the repositories that will be searched by the architecture. However they are indicative of the level of processing power required to index and search such large collections of classes. The tests for recall and precision will be run on a much smaller test repository that has been hand selected to test the search engine capabilities.

6.2 Indexing Speed

This test aims to show the initial time it takes to perform static indexing, though high, is acceptable. This we will show by testing the indexing mechanism on a set of test repositories of varying sizes and numbers of class files. The time taken to complete the indexing will be recorded and should give an indication of the prototypes indexing speed. The question of acceptability arises. This is a difficult definition to pin down as it will vary from user to user. We consider time in the order of minutes to be acceptable, but hours of indexing may

be considered unacceptable. This is justified by comparison to a users typical development task, such as compiling source code.

6.2.1 Method

For this test seven class repositories in .jar format were selected, ranging in size from 290KB to 22.5MB. The time to create a single index file was recorded for each repository. In this test we are interested in the computational cost of generating an index file from scratch. We would begin each test with no current index and a fresh JVM that had no classes still loaded. This shows feasibility in terms of indexing speed as it records the time taken from the moment a user actions the index, to the time the process completes.

6.2.2 Hypothesis

Ideal results in this test will show two properties; that there is a consistent proportionality between the cost to index and the size of the repository, and that the cost is reasonable in terms of usability. This will show that the pre-indexing process is both scalable and feasible.

6.2.3 Results

The results for this test are summarized in Table 6.1. They show a sample of class repositories that have been indexed. The size of each repository, and the number of classes contained in it, are presented, along with the time per 100 classes taken to create the index.

Name	Size(MBs)	# of Classes	Time/100 classes
jdk4.jar	22.5	7037	0.58
eclipse.jar	19.2	11768	0.72
jdk3.jar	10.5	5438	0.6
oz.jar	8.4	2442	0.69
jxta.jar	3.6	2226	0.73
db.jar	1.0	161	0.71
mjlib.jar	0.29	98	0.69
Average	9.36	4167	0.67

Table 6.1: Indexing Test Results

6.2.4 Interpretation

In Figure 6.1 we see that the results correspond almost perfectly to a linear relationship. The implications of this are discussed in the next chapter.

6.3 Searching Speed

This test will determine the average search times for varying size repositories. The aim of this test is to show the search times are reasonable and scalable. We regard a reasonable search to be one that would be timely for a developer who has provided the query and awaits the results. Tests that would also include the construction of a query would

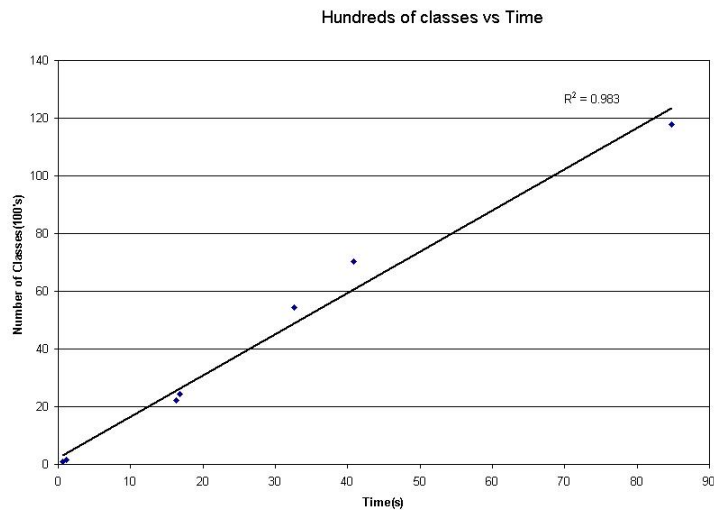


Figure 6.1: The linear relationship between number of classes and time. This demonstrates the scalability of the search to large repositories

require significant improvements in user interface and quantifiable usability studies. This test focuses exclusively on the search engine implementation, and how well it handles large repositories.

6.3.1 Method

This test runs four test case searches on seven test repositories to determine an average search time based on the repository size. The test cases were just simple input output examples such as string concatenation, object state change and numerical manipulation. The test cases were run on the same repositories used in the first test. This is less than desirable, because six of the seven repositories are not built with re-use in mind. Because the repositories do not all contain explicitly re-usable components the results may not be entirely indicative of actual search times. However given the large size of some of the repositories, we should be able to gauge an indication of the cost of actually performing a search. This will indicate the feasibility of such searches on a large scale.

6.3.2 Hypothesis

In this test we expect to see search times increasing with the size of each repository. This would be an indication of the higher cost of candidate execution that comes with behaviour searching.

6.3.3 Results

The results shown in Table 6.2 give an indication of the Search times for various repositories, unfortunately these results are not entirely expected, but valuable for other reasons, as outlined in the next chapter.

Name	Size(MBs)	# of Classes	Time/100 classes
jdk4.jar	22.5	7037	0.02
eclipse.jar	19.2	11768	*
jdk3.jar	10.5	5438	0.02
oz.jar	8.4	2442	0.01
jxta.jar	3.6	2226	0.02
db.jar	1.0	161	0.06
mjlib.jar	0.29	98	0.11
Average	9.36	4167	0.04

Table 6.2: Searching Test Results

6.4 Search Accuracy

The main purpose of this test is to show that the search engine is reliable. To show this we will run tests using hand selected sample behaviour. This behaviour will be used to search for the target methods. Our tests should show that the target method is found, and few (or no) other methods will be found erroneously.

6.4.1 Method

We will run six test cases, whose behavioural examples will be hand picked by inspection of the repository. The accuracy of the search will be recorded, indicating whether the component sought was found, and where additional components were found, whether they are relevant or not. We will also note any interesting findings during testing that arise.

This test will show both false-positives and false-negatives. False-positives are components found that do not implement the desired behaviour, but which our search algorithm has returned in the result set erroneously. Conversely false-negatives are suitable components that we know are in the repository, that are not returned by the search. These are harder to detect, but by hand selecting test cases, we should be able to detect the occurrences of false negatives. Accuracy is a key ingredient in feasibility, without low false positive rates and few false negatives, the search will be no more beneficial than text searching.

6.4.2 Hypothesis

In this test we should see the desired component being found in each case, and ideally there will be few or now false-positives suggested erroneously.

6.4.3 Results

The results are presented in Table 6.3 each test case with the number of components found and issues raised by the test case.

Test Case	Found	Issues
urlEncode	1	Parameter order
fullPath	1	Security
isVowel	2	Disambiguation
mooresLaw	1	
OctalEncode	1	
delete File	2	UI and Security

Table 6.3: Searching Test Results

Chapter 7

Discussion

In the previous chapter we presented the results of feasibility testing on a prototype of our architecture. In this chapter we analyze the meaning of those results. The first section of this chapter will look at the test findings, discussing the results and the issues that arose during testing. Section two will look at the wider implications of these test results, in terms of the applicability of this search technique, and supporting architecture, to the software development industry.

7.1 Findings

In this section we will discuss the results of each test in chapter six. The testing identified issues with the prototype we will look at these in terms of the search technique.

7.1.1 Index Speed Findings

Analysis of results

The results of the index speed testing were promising in terms of both feasibility and also scalability.

In terms of feasibility, the index cost was shown to be manageable even for large repositories. This is because the index cost is considered 'one-off' and is not an interactive task. This means it can be run in the background, and will not impede the progress of the developer as they will not be waiting on the indexing completion. Because the indexing of repositories will occur remotely, it is of no concern to a developer searching for components that indexing takes several seconds, as the indexing of remote repositories is unrelated to the local system.

Moreover, the cost to add small numbers of classes to an existing repository is very low, as illustrated in chapter six, the cost per 100 classes is under a second. This is much more favourable than similar tasks such as compilation, which for 100 classes will generally take longer than a second.

In terms of scalability, the test results showed that an increase in the number of classes indexed causes a proportional increase in the time taken to index. This proportionate rise in index time means that the indexing cost per 100 classes is a near constant value. The relationship between quantity of classes and time is linear.

Implications

There are several key observations to be made about the test results. The first is that the size in MB's of a repository is less indicative of the indexing time, than the number of classes in the repository. This is seen in the R^2 value of the graph in chapter six. This indicates that the size of each class is less of a factor in the indexing time than the actual loading of that class.

$$time_{index} = time_{load} + time_{extract}$$

$$time_{extract} = k_1 * size_{class}$$

$$time_{load} = k_2 + size_{class}$$

The equations presented above show a function of the index time (where k_n is some constant), being composed of the time it takes to load a class ($time_{load}$), and the time it takes to extract information from that class ($time_{extract}$). The time it takes to extract depends on the size of the class. The time it takes to load depends on the size of the class, and also some constant system overhead (for example, actually linking the class in the virtual machine). Given the results we can see that this constant time is much greater than the cost to extract information. This is shown by the fact that components with much larger average class sizes in MB, do not take proportionally more time to index than smaller ones. This means there is ample scope to increase the amount of information gathered at indexing time, without increasing the cost of the indexing process dramatically.

Lastly the time to index small repositories of 100-200 classes was very short, in the order of 1-2 seconds. This is interesting in terms of applicability, as repositories of this size would be roughly what an average user would be indexing on the network. This means that the repository index could be dynamic which would allow classes to be updated on the fly, meaning users who are searching a repository get the most up-to-date version of a component.

7.1.2 Search Speed Findings

This section presents analysis and observations of the search speed testing presented in chapter six.

Analysis of results

The results here were surprising, they were contrary to our initial hypothesis that larger repositories would take longer to search. The larger repositories were able to be searched faster than smaller ones in some test cases. This shows two properties of the search.

Firstly the index is performing well, because the time to determine suitability of components for execution is very low. The search engine can process classes that have low re-usability faster than those that are more reusable. Classes that are not suitable for invocation are filtered out based on information gathered from the index. This can be done quickly for code that has little or no re-usability. However when code is re-usable, then we must actually invoke the services and check the output. This process was shown to take a considerably larger portion of all searching time. The evidence for this is in the fact that the smallest, most re-usable component repository was the one which had the longest time per 100 classes. The time to invoke the additional services in classes that offer considerably more re-usable services is roughly twice as long. The time it takes to determine the suitability of a candidate for invocation is low, due to the index. However the time to invoke a component and determine the output compatibility is much longer. We see the time to search being

much higher in repositories that consist of many reusable services, because more services cannot be ruled out during filtering and must be invoked.

Secondly the search speed results are inaccurate because of the lack of chain testing. Test cases which utilized the chaining algorithm proved to be too unstable, and were avoided. As a result the search times will be skewed lower than they would otherwise be. This additional cost could be offset though. As outlined above there is ample opportunity to add more complexity to the indexing process, which could involve the addition of chain pre-calculation. This would keep the extra cost of this invocation to a minimum, and make full use of the additional speed the index affords.

Implications

The cost of invocation and comparison is relatively fixed. This means that for the search cost to improve we must be able to shift more of the dynamic burden away from the search engine and into the static index. This could be achieved by calculating short chains of candidates when they are added to a repository, so that the search engine need not perform these tests dynamically.

7.1.3 Search Execution Findings

This section presents analysis of the results of search accuracy testing, and will also look at several issues that arose during testing.

Analysis of results

The testing showed two important results, low false-positive rate and low false-negative rate. The low false-negative rate means that the search has sound accuracy, if we are trying to find specific behaviour that exists in the repository, we will definitely find it. The tests also show that the false-positive rate is very low. This means that the number of components that meet the required behaviour is often only one or two. This compares favourably with the text search outlined in the motivation chapter, where the percentage of false-positives for a keyword search was often in the 70-80% range. This shows that in addition to soundness of results, the search is also favourable in terms of accuracy of results over a keyword search.

Implications

This test more so than the previous two raised several important issues. These relate closely to limitations that were identified in previous chapters. We will briefly summarize what happened in the test, and how that relates to one of the limitations.

- *Parameter Order*

The permutation aspect of the search engine was shown to be working well, as it was able to find the desired service, irrespective of the order of parameters. The complexity of this algorithm increases in a factorial manner, so for services with greater than 6 parameters, execution times could expect to become much longer.

- *Disambiguation*

We have noticed that in some cases it is difficult to provide sufficient example behaviour to disambiguate components. It appears to occur when one behavioural representation lies within another. For example the `isAlphanumeric` function, is a superset of the `isVowel` function. Likewise members of the false result for `isVowel` are mostly false for `requiresEncoding`.

- *Security* One of the major issues raised with these test cases was the potential for dangerous execution. One of the services was simply a `deleteFile` function, which took a string. If we had been searching with the string `"/home/schrodashl/"` then my entire home directory would have been unknowingly wiped out. The solution appears to be to run tests in a second virtual machine that has limited security privileges on the real filesystem.
- *User Interface*
Components that require user interaction still require user interaction when invoked in this search, this would be problematic for users who are checking a large repository of code that prompts for user input. There are few ways around this unfortunately, unless we can automate answers to input questions.
- *Termination*
If we had solved this problem, it would not be appearing as a small note in paper on behavioural sampling. This is an undecidable problem in logic and computer science. The best solution we have to offer is the user specifying some finite upper bound on execution, and simply halting the execution after that time. This would make the search usable, but could potentially miss components that require lengthy computation.

We can see that despite being functional, the prototype is still in need of many more improvements before it could be considered robust enough for general purpose use. This notion of behavioural search applicability to the industry will be looked at in the next section.

7.2 The Wider Implications

In this section we analyze benefits and disadvantages that this behaviour searching technique offers. From this analysis we can determine the applicability of this technique to several phases in the software development life-cycle (SDLC).

7.2.1 Benefits

The search offers several benefits over and above its alternatives.

- *Accuracy*
The preliminary search results have shown promising results in terms of accuracy. The low false negative rate indicates the search is sound. The low false-positive rate means that the user will have less manual inspection to do during the reuse process. This compares favourably with the high false-positive rates associated with text-searching in chapter 2.
- *Query Difficulty*
The creation of input-output pairs is generally not difficult, as the desired behaviour of a component is known to the developer. This means that the user of this search technique should be able to construct a query with more ease than that required to create a formal specification.
- *Speed*
Results show that the speed of executing a search query has the same order of magnitude as other common developer tasks such as compilation. This should mean that it does not burden the developer who is using the search. Indexing is a slower task to

perform, but as outlined earlier, this slowdown is not a burden because indexing is a non-interactive background task.

7.2.2 Disadvantages

- *Instability*
The prototype is still unstable, many parts of it do not perform consistently and in its current state would be a hindrance to reuse. However with further development the technique could mature into a useful reuse tool.
- *Experimental Results*
The results gathered so far are initial and based on an unstable prototype. The quality of the test data that results from testing a prototype is not sufficiently high to allow us to draw any concrete conclusions, but the results do give promising early indications of both speed and accuracy.
- *Language Dependence*
The implementation of the architecture will, by the nature of the required meta-level API be language specific. This means that one implementation for Java, will be difficult to apply to any other language. This limits the applicability of the technique, dependent on the implementation languages available for it. Furthermore, if we integrate the querying into the language as well, then the queries themselves become language specific.

7.2.3 Applicability to the SDLC

The technique presented could be of benefit to several phases of the software development life-cycle (SDLC). In this section we will look at each of those sections, and identify phases that stand to gain from inclusion of tool-support for reuse.

- *Design*
Responsibility-driven Design [22] is a process which seeks to assign objects certain responsibilities which they must provide. This technique could allow early identification of desired behaviour, which would become part of a design. This behaviour specification can then be used to search for components early in the development life-cycle, making the implementation phase shorter.
- *Implementation*
During implementation, the understanding of required behaviour at the early stages could make finding components that provide close matching behaviour a reality. This would allow components that perform similar functions to be modified for the particular situation in which they are needed.
- *Corrective Maintenance*
One of the generally accepted benefits of re-use is the higher quality of the code. This means that the debugging has been carried out previously, and so the burden on the user of pre-existing components is lower. With the network providing a sense of community, users could get rated on the quality of the components they share, so that high rated users code is known to be of higher quality.
- *Evolution*
The evolution and perfective maintenance of reusable components can be an issue in systems. If a component is upgraded by its author, the bug fixes and improvements

will not apply to a component in use. However with the index and network capabilities of this architecture, there is scope for notifying users who have integrated a component into a system, that the component has been updated and they should get a new version.

The prototype architecture could be of significant use to the software development life-cycle given the benefits it could provide. However the current limitations with the prototype mean that any further testing for feasibility would require further development of the prototype.

Chapter 8

Summary

8.1 Future Work

This project has developed an improved search technique that accommodates the complex behaviour of object-oriented components. This search has been incorporated in an architecture that provides tool support in such a way as to alleviate issues with speed and scalability that are present in older behaviour based searches. This architecture has been implemented and the prototype has been tested. This has shown promising feasibility results.

In this section we turn our attention to future developments, and areas where the project can be advanced. This can be achieved in two ways:

- Improve the architecture and prototype
- Apply the technique to other software engineering research

The next two sections will present future places for development and advancement related to the above categories.

8.1.1 Areas of Improvement

The prototype is far from complete, several of the core modules are not entirely implemented, and parts that are implemented are not completely bug-free. There is much work that could be done improving the prototype, or the design. The implementation could be improved by addressing limitations in the modules such as:

- Index could be upgraded to relational database.
- Search engine could have several bugs relating to the recursive chaining dealt with.
- Set-up and tear-down aspects could expand the range of components describable in the querying language.
- The Network connectivity could be plugged into the current stub.
- The GUI could be integrated into the runtime environment, in a similar way to Squeaks' methodFind.

These improvements would help to make the tool support more reliable, and indeed improve the accuracy and speed of the search. This will lead to improved feasibility and potentially applicability to the real world.

8.1.2 Areas of application

The tool could be applied to several other research areas of computer science with little or no adaptation. This would demonstrate the applicability of this technique in fields of research far removed from software reuse. In this section we will discuss two such areas of application, and how the use of this technique may prove beneficial to these areas.

In the field of research concerned with agile software development, XP (extreme programming) [2] promotes the creation of test cases before code development. These test cases are, in effect, programming language based behavioural specifications. They could be modified and sent to the search engine. If this were implemented, then perhaps extreme programming developers would be able to avoid coding certain functionality at all, choosing instead to use components discovered by the behavioural search engine.

Some software researchers are seeking ways to help users of large complex API's find their way around [12]. Our tool would be beneficial here as in some cases it could augment current techniques, to disambiguate multiple paths. In other cases it could be a way of finding code snippets within source code repositories, that perform certain functionality, without having to know what that functionality is commonly called.

Lastly the peer-to-peer aspect of this tool could be extended to include a community of developers. This close interaction between developer could foster a positive co-operation that leads to high levels of re-use and re-use focused development. It would certainly help open source project teams share domain-specific code between themselves.

8.2 Contributions

From the outset we aimed to achieve three contributions

- Develop an object-oriented Behavioural search that allows for the varied way in which objects can create output.
- Design an architecture that uses the new search, in a way that helps to alleviate some of the other difficulties with BS
- Lastly we wanted to implement and test this architecture, to determine if it is feasible.

All three of these contributions have been achieved.

8.3 Conclusion

This project has presented a technique for behaviour-based searching, and incorporated this in a tool-support architecture. This searching technique reduces the cost of re-use by presenting the user with a set of results which can be shown to include only components that exhibit to some extent the desired behaviour. We presented our initial implementation of this architecture, called codeCollective, which though limited was able to show that indexing and searching times are acceptable, and that the recall of tests is consistent with what we expect.

Bibliography

- [1] ATKINSON, S., AND DUKE, R. Behavioral retrieval from class libraries. *Australian Computer Science Communications* 17, 1 (1995).
- [2] BECK, K. *Extreme Programming Explained*. Addison Wesley, 2000.
- [3] GROUP., T. X. L. R. *The Smalltalk-80 System*. BYTE Magazine, 1981.
- [4] GUILLAUME BRAT, RICH WASHINGTON, D. D., AND GIANNAKOPOULOU, D. Experimental evaluation of verification and validation tools on martian rover software. In *Formal Methods and Design* (2004), pp. 167–198.
- [5] HALL, R. J. Generalized behavior-based retrieval. *Proceedings of the 15th international conference on Software Engineering* (1993).
- [6] HELM, R., AND MAAREK, Y. S. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 1991), ACM Press, pp. 47–61.
- [7] HEMER, D., AND LINDSAY, P. Supporting component-based reuse in care. In *CRPITS '02: Proceedings of the twenty-fifth Australasian conference on Computer science* (Darlinghurst, Australia, Australia, 2002), Australian Computer Society, Inc., pp. 95–104.
- [8] INGALLS, D., KAEHLER, T., MALONEY, J., WALLACE, S., AND KAY, A. Back to the future: the story of squeak, a practical smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1997), ACM Press, pp. 318–326.
- [9] JENG, J.-J., AND CHENG, B. H. C. Specification matching for software reuse: a foundation. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability* (New York, NY, USA, 1995), ACM Press, pp. 97–105.
- [10] KRUEGER, C. W. Software reuse. *ACM Comput. Surv.* 24, 2 (1992), 131–183.
- [11] LIM, W. C. Effects of reuse on quality, productivity, and economics. *IEEE Softw.* 11, 5 (1994), 23–30.
- [12] MANDELIN, D., XU, L., BODIK, R., AND KIMELMAN, D. Jungloid mining: helping to navigate the api jungle. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM Press, pp. 48–61.
- [13] MCILROY, M. Mass produced software components. *Report on the NATO Software Engineering Conference* (1968), 79–87.

- [14] MILI, R., MILI, A., AND MITTERMEIR, R. T. Storing and retrieving software components: A refinement based system. *IEEE Trans. Softw. Eng.* 23, 7 (1997), 445–460.
- [15] NIU, H., AND PARK, Y. An execution-based retrieval of object-oriented components. In *ACM-SE 37: Proceedings of the 37th annual Southeast regional conference (CD-ROM)* (New York, NY, USA, 1999), ACM Press, p. 18.
- [16] PODGURSKI, A., AND PIERCE, L. Retrieving reusable software by sampling behavior. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (1993).
- [17] PRIETO-DIAZ, R. Implementing faceted classification for software reuse. *Commun. ACM* 34, 5 (1991), 88–97.
- [18] PRIETO-DIAZ, R. Status report: Software reusability. *IEEE Softw.* 10, 3 (1993), 61–66.
- [19] SALTON, G. Another look at automatic text-retrieval systems. *Commun. ACM* 29, 7 (1986), 648–656.
- [20] TRAAS, V., AND VAN HILLEGERSBERG, J. The software component market on the internet current status and conditions for growth. *SIGSOFT Softw. Eng. Notes* 25, 1 (2000), 114.
- [21] VITHARANA, P., ZAHEDI, F. M., AND JAIN, H. Design, retrieval, and assembly in component-based software development. *Commun. ACM* 46, 11 (2003), 97–102.
- [22] WIRFS-BROCK, R., AND WILKERSON, B. Object-oriented design: a responsibility-driven approach. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications* (1989), ACM Press.
- [23] ZAREMSKI, A. M., AND WING, J. M. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.* 6, 4 (1997), 333–369.