Web Intelligence and Agent Systems: An international journal 6 (2008) 0
IOS Press

Knowledge Reuse for Software Reuse

Frank McCarey*, Mel Ó Cinnéide and Nicholas Kushmerick
School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland.

Abstract. Software reuse can provide significant improvements in software productivity and quality whilst reducing development costs. Expressing software reuse intentions can be difficult though. A developer may aspire to reuse a software component but experience difficulty expressing their reuse intentions in a manner that is compatible with, or understood by, the component retrieval system. Various intelligent retrieval techniques have been developed that assist a developer in locating or discovering components in an efficient manner. These solutions share a common shortcoming: the developer must be capable of anticipating all reuse opportunities and initiating the retrieval process. There is a need for a comprehensive technique that not only assists with retrievals but that can also identify reuse opportunities.
This paper advocates that component-based reuse can be supported through knowledge collaboration. Often programming tasks and solutions are replicated; this characteristic of software can be exploited for the benefit of future developments. Through the mining of existing source code solutions, knowledge, relating to how components are used by developers, can be extracted. Based on a developer's current programming task, this knowledge can subsequently be filtered and used to recommend a candidate set of reusable components. This novel recommendation approach applies and extends commonly used Information Retrieval and Information Filtering techniques such as Collaborative Filtering, Content-Based Filtering, and Bayesian Clustering Models, to the software reuse domain. This recommendation technology is applied to several thousand open-source Java classes. The most effective recommendation algorithm produces recommendations of a high quality at a low cost.

Keywords: Component-Based Development, Knowledge Collaboration, Recommender Agent, Information Filtering.

1. Introduction

Software reuse has been practised ever since the advent of programming itself, albeit in a somewhat ad-hoc manner [15,34]. Reuse enables developers to leverage past accomplishments and can facilitate significant improvements in software quality [51]. Additionally, reuse has been shown to improve productivity [26], shorten the development cycle [7], and reduce development costs [43]. These advantages, combined with smaller development budgets and the increased availability of reusable artefacts, are just some of the reasons why software reuse has become more prominent of late.

Despite the many benefits, the software reuse community has failed in its attempts to establish reuse as a standard software engineering practice. This is evident from the lack of standard reuse practices even though key reuse research dates back 40 years [34]. This failure can be attributed to numerous factors such as the lack of a standardised market place for reusable software artefacts and the inadequacy of reuse support tools.

Candidates for reuse include source code, software architectures, and support documentation. The focus of this research is limited to source code reuse through software components. In component-based reuse, new applications are composed of many small reusable software components, typically stored in a component library. Components must be accessible, reliable, and of high quality; this paper concentrates on improving component accessibility. We have identified three underlying related problems that restrict component reuse, these are: the inability of support tools

*Corresponding author. E-mail:frank.mccarey@ucd.ie Phone:+353-87-7949237 Fax:+353-1-2697262

to automatically identify reuse opportunities, the separation of reuse from mainstream development activities, and the lack of techniques to store and subsequently distribute task-relevant component knowledge among developers. Combined, these act as a major barrier to component reuse and are the central motivation for this research.

Effective tool support is essential for promoting software component reuse as component libraries tend to be large and growing. For example, the latest version of the Java API library has over 3000 classes while the Java *Swing* library contains more than 500 classes. In addition to commercial components, a mature software development organisation could maintain a large library of in-house components. Conversancy with all components in a large library is practically impossible. As a result, there are many stored components that developers are unaware of and, accordingly, that they never make any attempt to reuse.

Typical component support techniques include library browsing *agents* [6], *web-based* search approaches [16], and component *ranking* tools [17]. These are described in section 7. Each solution attempts to assist developers in discovering and locating components in which they are interested. A common shortcoming of these solutions is that reuse is viewed as a separate activity from software development; a developer must halt their current programming task, initiate the component search process, and on completion, return to their program task. As previously noted, developers are not aware of all available methods in a library or may be unable to express their reuse intentions clearly. If they believe a reusable component for a particular task does not exist then it is less likely that they will initiate a component search. Reuse support tools that focus solely on search techniques are insufficient as they are only useful when a developer makes a reuse attempt.

When such retrieval tools prove inadequate, a developer will attempt to acquire knowledge relating to components from peers. Typically a software component is used by many developers in many applications. When a developer is attempting to solve a problem or make a reuse attempt, they seek the assistance of developers who have solved this or a similar problem previously through the use of software components. This could involve direct meetings, email, or telephone exchanges between colleagues. Such approaches may be effective but are clearly an unreliable and inefficient means of sharing knowledge [23].

This research shifts the focus from component retrieval tools to a component recommendation technique that simplifies knowledge collaboration. A methodology is presented in this paper that can automatically identify reuse opportunities and, accordingly, recommend a candidate set of reusable components to a developer. Recommendations are intelligent, relevant, and of value to individual developers. This methodology capitalises on the similarities that exist between different source code solutions. Programming tasks, and consequently solutions, are often mirrored inside an organisation, across a community, or within a specific domain. This explains the phenomenon of source code clones and code duplication [35]. From mining existing source code repositories, a collective collaborative knowledge-base can be established that can be used to support and predict future component reuse. At present, much of the knowledge ingrained in source code is never captured.

For each reusable component, information relating to where that component is used, how it is used, and how it integrates with other components, can all be collected through mining source code repositories. The methodology presented in this work uses an intelligent software agent that employs this collaborative knowledge-base, along with advance filtering algorithms, to identify and to recommend component reuse opportunities for a developer. Such recommendations are based on a developer's current programming task. Knowledge-based component recommendation improves and complements traditional search-based component retrieval schemes. Our methodology is proactive, requires no additional effort from developers, and easily allows component knowledge to be shared among developers.

The following are the principal contributions of this research:

– *The application, mergence, and extension of several Information Retrieval and Information Filtering schemes to a software reuse domain.* Information Retrieval models are used with both conventional and novel filtering algorithms to generate component recommendations. Using customary and specialised retrieval metrics, extensive evaluations have concluded that highly relevant component recommendations can be produced, for a pro-

grammer, using these algorithms. In particular, the new algorithms, that consider ordering information, produce encouraging results. Generally, these positive results affirm the information-rich characteristic of source code.
– *A lightweight knowledge-acquisition and knowledge-sharing methodology that increases the accessibility of software components.* The validity of this agent-based knowledge sharing methodology is demonstrated using a prototype software implementation that is applied with full rigour to thousands of Java methods.
– *A novel, proactive approach to software reuse support.* Unlike previous approaches to this problem, our approach is intelligent, proactive, and autonomous. Additionally, this economical solution can easily be adopted: the only prerequisite is the availability of a source code repository.

The remainder of this paper is organised as follows. The next section presents an overview of software reuse and Component-Based Development. This is followed by a discussion on component reuse from a developer's perspective in section 3. Section 4 describes our prototype component recommender tool, named RASCAL, which employs intelligent agent technology. Component recommendation algorithms are fully explained in section 5 and recommendation results are reported in section 6. Related works are reviewed in section 7 and, finally, conclusions are drawn in section 8.

2. Software Reuse and Component-Based Development

Software reuse is an umbrella concept that has been around for many years. The term was originally coined by McIlroy [34] at the NATO conference on Software Engineering in 1968. Generally software reuse can be defined as the process of creating software systems from existing software rather than building software systems from scratch [21]. In addition, it can also be defined as reusing existing software artefacts during the process of building a new software system. A software artefact may be a tangible item such as source code or a test case. An artefact can also refer to a piece of formalised knowledge that can contribute, and be applied, to the software development process [8]. Typical reusable artefacts include product families, design architectures, and documentation.

This research is limited to source code reuse through software components. *Component-Based Development* (CBD) is an emerging software development paradigm, promising many benefits including reduced development and maintenance costs, and increased productivity. CBD involves building applications using prefabricated software components. No standard definition exists of software components, however, one widely accepted definition is: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. An interface is a set of named operations that can be invoked by clients. Context dependencies are specifications of what the deployment environment needs to provide, such that the components can function." [53]

Components help to reduce the time required to design, implement, and debug new software while the cost of developing reusable components is offset by repeated use. Popular examples of reusable component libraries include the Sun *Swing* library and the Visual Numerics *International Mathematical and Statistical Libraries* (IMSL).

Creating reusable components and component libraries can be difficult, time-consuming, and expensive. However, positive progress has been made in recent years; for example components can be sourced commercially *off-the-shelf*, from repositories of *open-source* software, or built in-house as part of *product families*. As noted in the introduction, the principal focus of this research is on increasing the accessability of stored components rather than populating component libraries.

2.1. Component Retrieval Techniques

This subsection describes several general classes of retrieval schemes; a complete review of principal related works in this area is provided in section 7.

Information Retrieval (**IR**) These approaches typically rely on class or method names, source code comments, and documentation [28]. The reuse query must be expressed in a natural language; this is desirable but it is often difficult to formulate queries in a way that ensures irrelevant components are not retrieved. Such techniques also fail to take into account a developer's current programming context, and thus, often prove inadequate.

Descriptive Whereas IR techniques represent components by their source text, descriptive methods rely on abstract descriptions of components. Examples include sets of keywords and facets. A limitation of this approach is that it is difficult to implement retrospectively, requiring greater upfront planning.

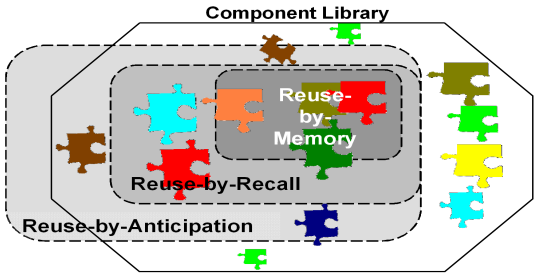
Syntactical The most common syntactical retrievals are based on signature matching [27, 58]. This approach is useful for quickly ruling out components that do not match a developer's reuse query. Syntactical approaches are more formal than IR and descriptive methods, and thus, are often challenging for a developer to use. Additionally, signature alone does not guarantee the expected behaviour of a component.

Formal Methods In this approach, a component is represented using a formal specification language. Likewise, queries are also submitted in a formal language. Correctness proving systems [61,19] or specification refinement systems [38] are then used to compare stored components with the user query. Typically, formal methods are highly accurate at retrieving relevant components but often prove difficult for developers to use.

Artificial Intelligence (AI) AI [47] schemes attempt to be smarter than the above traditional methods, in that they endeavor to understand both the library components and the reuse query. These approaches tend to be context and environment aware, and generate retrievals based on this. AI techniques typically require a knowledge-base but this cost is counterbalanced against effective retrievals.

Web Technologies Search and retrieval techniques usually applied to web-based searches can also be applied to component retrieval. The advantages of such approaches include scalability and efficiency while problems including security and legal concerns [16].

The goal of this research is to develop a new component retrieval methodology that builds and improves on existing techniques. The limitations of traditional IR, descriptive, and syntactical approaches have been noted: each technique is unreliable, require up-front planning, and lack context information. Formal methods are beneficial for proving retrieval correctness but are often difficult for developers to use; perhaps the lack of recent research in this area can be attributed to this. More promising are web-based and knowledge-based AI retrieval approaches. Retrieval alone, though, is not sufficient to assist a developer with component reuse.



Concept Taken From - Ye, Y. and Fischer, G.: 2002, Information Delivery in Support of Learning Reusable Software Components on Demand, *in Proceedings of the 7th International Conference on Intelligent User Interfaces*, ACM Press, San Francisco, USA, 2002

Fig. 1. How Developers Reuse

3. Component Reuse: The Developer's Perspective

The previous section presented several different retrieval techniques. The principal goal of such techniques is to facilitate and encourage reuse. However, any competent component reuse support tool that employs these retrieval methods must also consider reuse from the developer's perspective. If a developer is to reuse a component, they must be able to locate that component with maximum ease, in minimal time. Frakes and Fox [10, 11] discovered that *no-attempt-to-reuse* accounted for 24% of software reuse failures. This strengthens the argument that a reuse programme cannot succeed without the backing of the development team, and thus, the human factors that hamper reuse need to be addressed.

3.1. Reuse: A Problem Solving Activity

Mili et al. [36] describe reuse-based development as a problem solving activity, the problem being that of finding a software component that satisfies a set of user requirements. From a component reuse perspective, Ye [57] categorises this activity into three separate tasks: *reuse-by-memory*, *reuse-by-recall*, and *reuse-by-anticipation* as is shown in figure 1.

Reuse-by-Memory Developers recognise similarities between their current development and programs that they have previously created or with components that they have used. This simplifies reuse as the developer can express their reuse intentions clearly.

Reuse-by-Recall Developers vaguely remember components in the library that they believe loosely matches their requirements. They may not be exactly sure which components they are or where they are located and thus retrieval tools are important.

Reuse-by-Anticipation Developers are not aware of any components that match their reuse needs. However, based on their knowledge of the component library, they anticipate that a component exists that satisfies their reuse requirements.

Reuse-by-memory and *reuse-by-recall* are reasonably straightforward and can be supported well through the component library browsing and retrieval tools previously described. *Reuse-by-anticipation* is somewhat more complicated:

1. A developer may incorrectly anticipate a component that does not exist.
2. It is difficult for a developer to clearly express their reuse intentions.
3. A developer cannot easily evaluate retrieved components due to their limited knowledge.

Assuming a retrieval technology is in place that overcomes these challenges, there still remains one major barrier to reuse: a developer may not anticipate the availability of a component which they can reuse in their current development. Hence, *no-attempt-to-reuse* is made. In such a situation, the available retrieval tools are merely passive devices because they become only useful when a developer decides to make a reuse attempt by anticipating the existence of certain components. The problem is further exacerbated by the likelihood of the component library increasing in size over time. Therefore, an intelligent support tool is required that can notify a developer of what components exist and when it is appropriate to reuse them.

3.2. From Development-with-Reuse to Reuse-within-Development

This subsection shifts the focus from the cognitive barriers to reuse, to the technical challenges

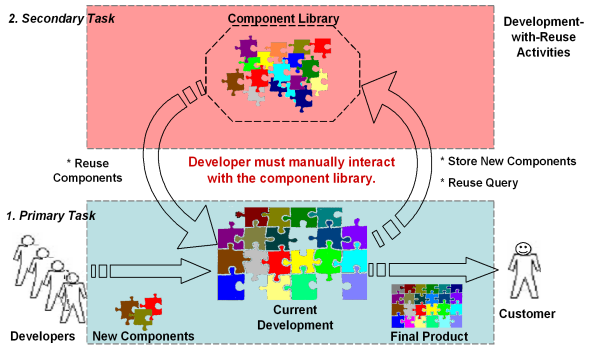


Fig. 2. Development-With-Reuse

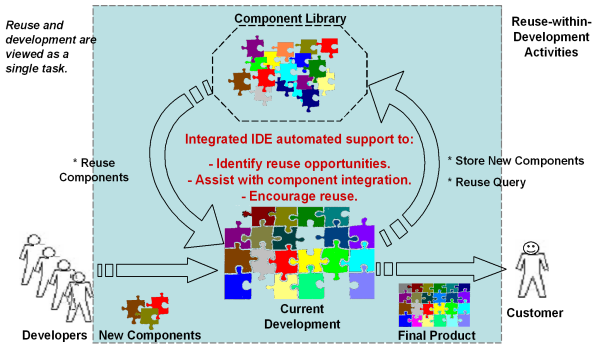


Fig. 3. Reuse-Within-Development

that impair reuse in practice. Traditional component library tools make little effort to address the *no-attempt-to-reuse* problem. They explicitly assume that a developer will, at the very least, search for a component even if they are unsure that a component exists that fits their requirements. Such component library systems are designed to support *development-with-reuse* [45].

The *development-with-reuse* paradigm, as shown in figure 2, views reuse as a stand-alone activity, independent of traditional programming activities and development tools. It supposes that developers want to reuse, and that they can easily express their reuse intentions and correctly formulate reuse queries. By distinguishing between normal development tasks and reuse tasks, there is an overhead of task switching between the two. Further to this, the tools used by developers in normal development typically differ from those used in the reuse process. Reuse must be well planned in advance for *development-with-reuse* to be successful. However, reuse can not be completely planned beforehand, it happens within the context and process of development [49]. Additionally, the Li et al. [25]

study has shown that reuse is often opportunistic, with components frequently not being selected until the detailed design and coding stage. Thus, a development paradigm shift is needed.

Reuse-within-development is a user-centered approach where the focus is on the behaviour and actions of developers, and the aim is to merge Component-Based Development activities with the existing practices of developers [18,37, 58]. This is illustrated in figure 3. *Reuse-within-development* differs from the methodology-centered approach of *development-with-reuse* which attempts to change and adapt a programmer's actions to fit the reuse model. In the user-centered approach, the development environment must provide support that: can automatically identify and retrieve relevant reusable components; assists with component understanding and integration; does not interfere with a developer's regular programming activities; and reduces the cost of component reuse.

By assisting with component understanding and integration, the development environment is not simply providing a component reminder service but rather it is facilitating component reuse with minimal ease. For example, a set of relevant code examples could be presented to a developer that quickly allows her to understand and integrate a component [58]. In addition, automatically locating relevant software components and delivering these to a developer ensures that developers need only be aware of a much smaller set of components. This enables greater component comprehension. In the next section, we describe RASCAL, a prototype tool that supports *reuse-within-development* in practice.

4. Knowledge Reuse: The RASCAL Component Recommender

This section introduces a knowledge-based agent that can automatically identify component reuse opportunities. After this, we describe our prototype component recommender tool, named RASCAL, that employs this agent technology.

4.1. Knowledge Reuse with Software Agents

Software development is a knowledge-intensive activity. Often, developers lack the specific knowledge that is required for a particular development task. In this scenario, developers typically seek assistance from their peers. LaToza et al. [23] have completed a study at *Microsoft* to accurately asses the impact of this knowledge deficiency problem; it was discovered that 62% of developers had serious problems due to switching tasks because their team-mates requested assistance, and 50% had serious problems having to switch tasks because they were unable to solve their current task without seeking assistance from team-mates or additional resources. This problem will extend to a Component-Based Development environment where component libraries are large and integration is difficult.

The tools and techniques used to share knowledge within organisations can vary greatly. As noted previously, colleagues can hold informal meetings, telephone or email each other, or perhaps rely on extensive support materials. These techniques are often effective, but lack efficiency and reliability. Therefore, we propose an intelligent software agent that can automatically share knowledge among software developers. This is similar to the work of Lodi et al. [4] where agents are employed, in a distributed environment, to discover and share knowledge. In the context of our work, the agent will shared knowledge relating to software components by identifying opportunities for component reuse.

Software Agents [1,47], like software components, are loosely defined. Agents differ from conventional software in that they are long-lived, semi-autonomous, proactive, and adaptive. Agents carry out some set of operations on behalf of a user, or another program, with a certain degree of independence. In so doing, the agent employs some knowledge or representation of the user's goals and desires. This work is concerned with the use of intelligent agent technology in the CBD domain; such agents will assist a developer to complete their development and component reuse activities in a way that is consistent with the fundamental principles of the *reuse-within-development* concept. To do this, the agent must first learn knowledge pertaining to software components.

4.1.1. Learning Knowledge from Source Code Repositories

When undertaking a development task, programmers often have a sense of déjà vu [35]. They

feel that they have solved this problem before or know of an existing solution. In general, medium and low level programming tasks are repeated: either within a project, an organisation, a domain, or within a community. Similarly, solutions often mimic each other. This explains the phenomenon of source code clones and code duplication. Developers recognise that a solution exists to their problem and decide to copy existing source code.

Instead of simply copying source code, embedded knowledge can be extracted or learned from the source code and reused in new projects. The knowledge collected will relate to how components are used by developers in practice, and will act as knowledge-base for the intelligent software agent. Subsequently, the agent will be capable of advising a developer, based on previous developers solutions, on how best to complete their current programming task using software components.

4.1.2. Sharing Knowledge with Developers

Just as it is important for the agent to mine and learn knowledge, it is also important that the agent can distribute and share this knowledge with the relevant parties. The agent must initially determine when knowledge is deemed relevant to a developer, and accordingly, must present this knowledge in a manner that is effective yet unobtrusive. Often the agent is offering unsolicited advice. This research concentrates on supplying developers with knowledge that assists component reuse; this typically involves recommending a set of components for reuse to a developer. The agent must: be easy to use, anticipate reuse opportunities, allow a developer to quickly assess retrieved components, and bridge the gap between a developer's reuse requirements and their actual reuse levels. The following section describes *RASCAL*, a component reuse support tool that employs an intelligent agent to produce recommendations.

4.2. RASCAL

RASCAL, as shown in figure 4, is a CBD prototype tool that is designed to support *reuse-within-development*. This tool incorporates intelligent agent technology and supplies a developer with task relevant component recommendations. The main components of RASCAL are now detailed.

4.2.1. Recommender Agent

The *recommender agent* is responsible for actively monitoring the current developer's actions. Based on this, the recommender agent formulates a reuse intention and proactively recommends a candidate set of ordered library components to the active developer. Recommendations are produced using the information filtering techniques explained in section 5.

The agent does not attempt to understand the functionality of the source code that a developer is coding or the developer's ultimate goals. Rather, the agent is intended to provide short-term and immediate assistance to developers. Likewise, the agent does not understand the functionality of the components that it recommends; a component is recommended simply because the agent believes that it is appropriate for a particular developer to use this component now. An agent learns such knowledge from mining source code repositories as discussed shortly.

4.2.2. Knowledge-intensive IDE (KIDE)

An *Integrated Development Environment* is used by developers on a daily basis to complete their programming tasks. RASCAL extends a developer's current IDE to incorporate the recommender agent, so that the agent and the IDE appear as a single entity to the developer. This is referred to as a *Knowledge-intensive Integrated Development Environment* (KIDE), and supports a *reuse-within-development* paradigm.

An example of the RASCAL KIDE is shown in figure 5. The screenshot of the *Eclipse* IDE resembles the standard *Eclipse Java Perspective* with the exception of the, unobtrusive, recommender agent that is visible at the bottom right corner of the window. As a developer is writing code, RASCAL monitors the developer's source code and employs this information to recommend a set of library components. The developer does not need to explicitly accept nor reject recommendations.

4.2.3. Source Code Repository

The *source code repository* can contain source code from: existing work, previous projects, and external software. Additionally, it is beneficial if the code repository contains the work of a heterogeneous set of developers, who are likely to have made use of the majority of library components. The open-source development paradigm is

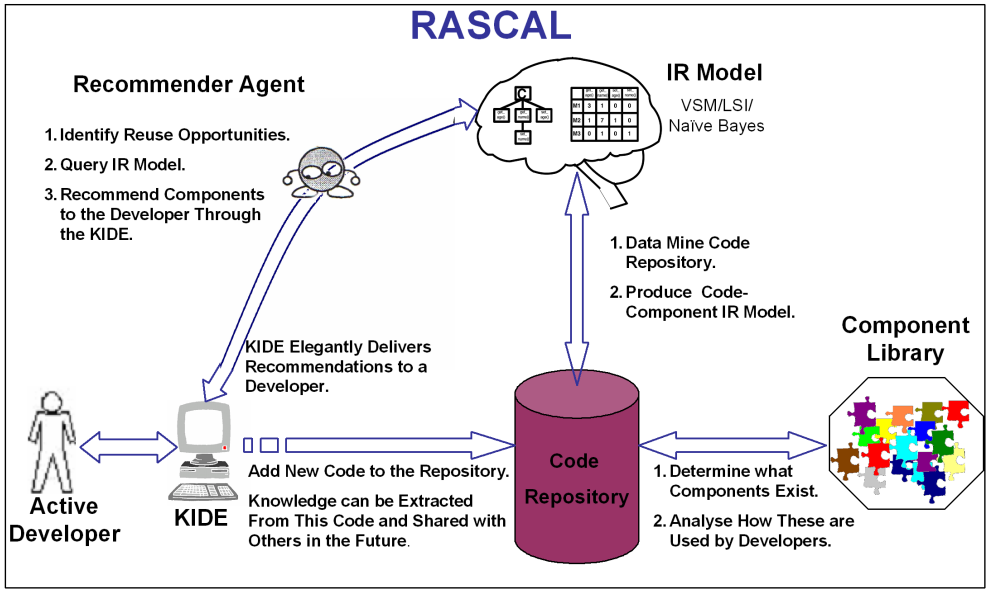


Fig. 4. The Main Components of RASCAL

well suited in this context with significant amounts of code available; accordingly, we make use of the Sourceforge [40] repository. The code repository must be sufficiently populated to allow effective learning. A large source code repository will likely contain information about many components, which in turn increases the number of components for which recommendation is possible, but could result in components which have a high density-of-use being recommend more frequently. With regard to the effectiveness of RASCAL, our previous research has shown that a small populated repository of source code that uses software components frequently is more desirable than a large populated repository of source code that infrequently employs components [33]. Ideally, the code in the repository will be of a high quality, having been thoroughly tested.

4.2.4. Information Retrieval (IR) Model

An *IR model* is populated from mining the source code repository. This represents a lot of the information contained in the code repository in a useful manner. Possible IR models are discussed further in the next section. The recommender agent queries this model and, resultantly, recommends a set of library components to the developer. Before the IR model can be built, information needs to be extracted from the code repository. Source code parsers, such as the *ByteCode Engineering Library* (BCEL)[5] used in this research, can be employed for this purpose.

4.2.5. Active Developer

For each individual *active developer*, RASCAL continuously recommends a relevant set of library components for reuse. The recommendation set contains components that a developer may or may not have anticipated. RASCAL only considers the content of a developer's current work and does not use any historical developer information; for example, RASCAL does not examine any programs that a developer has coded in the past. Although a developer may have made use of certain components in the past, this does not guarantee that the developer will recall, or identify appropriate opportunities to reuse, all such components. Therefore, RASCAL merely considers the program under development. The developer's current component usage information is stored along with ordering details, as shown in table 1. This latter ordering information is used by our *Content-Based Filtering* algorithm, as explained in section 5.3.

4.2.6. Component Library

Component libraries can be populated through a variety of sources; for example, from built in-house components, open-source software, or from commercial components. This research focuses on

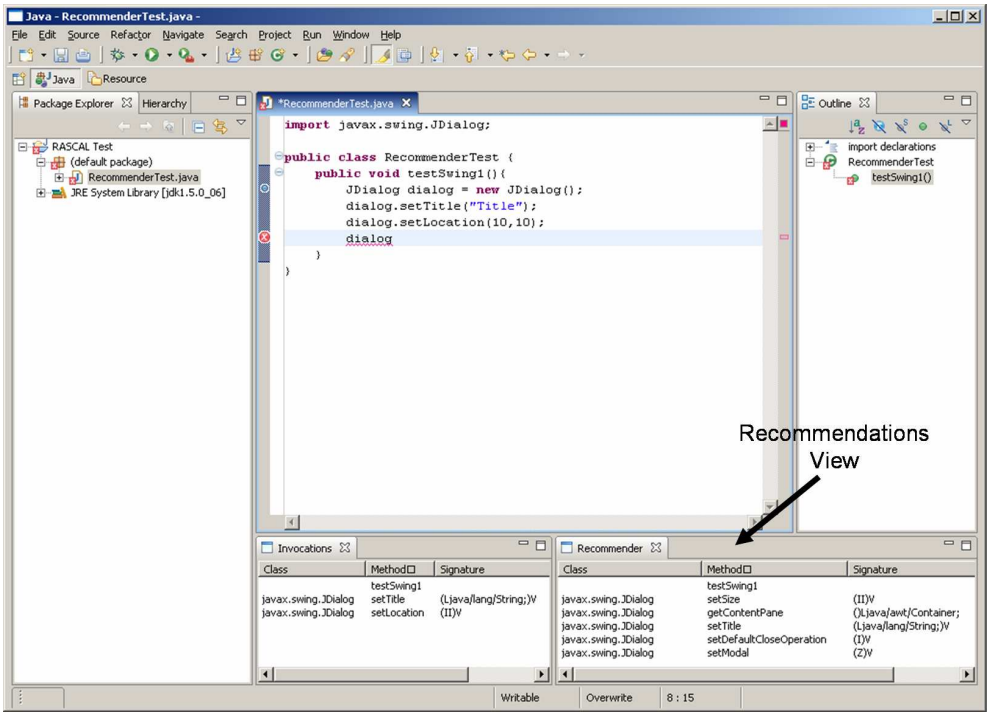


Fig. 5. RASCAL Eclipse Plugin

Table 1
Active Developer's Component Vector

Component Method	User: U1
JLabel:setBorder(Border)	1
JFrame:setDefaultCloseOperation(int)	1
JPanel:add(JLabel)	1
JPanel:add(JLabel)	1
JPanel:add(JLabel)	1
JFrame:setVisible(boolean)	1

improving the accessability of components in a library rather than populating component libraries.

Two important components of RASCAL are the *recommender agent* and the *Information Retrieval model*. The next section explains the technical aspects of both.

5. Component Recommendation

The previous section introduced a *Knowledge-intensive Integrated Development Environment* prototype tool, named RASCAL, that incorporates intelligent agent recommendation technology. This section presents the filtering methodology employed by the recommender agent and, to a lesser extent, reviews a number of IR models. The IR models employed in this research are standard techniques and thus deserve minimal explanation.

5.1. Granularity

This subsection clarifies the granularity at which RASCAL operates by clearly defining three terms that are commonly used in recommender literature.

Item RASCAL recommends an *item* to a developer. Specifically, RASCAL recommends a reusable Java method from a component library to a developer. Developers are likely to use several items within their programming solution. When statically comparing two pieces of source code, the items (methods) invoked within each source code are compared. During comparison, method signature is ignored; library methods overloaded within a particular class are treated as a single library item. Typically, overloaded methods perform similar operations but operate on different input data. The recommendation methodology presented here does not distinguish between overloaded member methods. Additionally, an object's precise type is not always identifiable; for example, in certain circumstances the runtime type of a polymorphic object may not be known without per-

forming a runtime analysis, thus the exact item invoked cannot always be statically determined. In this scenario, the corresponding item in the parent class is recorded as the invoked item.

Active User A *user* is a Java method, written for a previous project, that is stored in the source code repository and invokes items. By examining many users, knowledge about the component library can be learned. The *active user* is the method currently being programmed by a developer. The developer, actively working on this method, can also be considered as the *active user*. Dependent on the context of use, it will be apparent when the term *active user* refers to a method and when it refers to the developer of that method.

Vote This represents a user's preference for a particular item. In this context, a vote is simply an invocation count for a library method.

Notably, the term user and item can both refer to Java methods but the distinction should be clear. Based on the active user's (developer) vote, over several items (library methods), a recommendation set of predicted votes can be produced using the following algorithms.

5.2. *Collaborative Filtering (CF) Recommendations*

The goal of a *Collaborative Filtering* (CF) algorithm is to suggest new items, or to predict the utility of a certain item, for a particular user based on the user's previous preferences and on the opinions of other like-minded users [48]. CF algorithms are used in mainstream recommender systems like *Movielens* [46]. These systems are founded on the belief that users can be clustered. Users in a cluster share preferences and dislikes for particular items, and are likely to agree on future items. As just noted, a user refers to the developer of a user-defined Java method and an item refers to a reusable Java method from the component library that can be invoked. Breese et al. [2] classified CF algorithms into two main classes, *Memory-based* and *Model-based*; both are explained here.

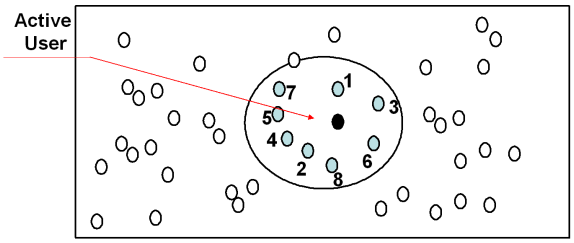


Fig. 6. Illustration of (kNN) Formation: After comparing the active user with 43 other users, a recommendation is produced based on the k=8 most similar source codes.

5.2.1. *Memory-based Collaborative Filtering*

In a *Memory-based CF* approach, a recommendation for the active user is derived by considering all other users in the code repository. Vote v_{aj} corresponds to the vote by user a for item j. The predicted vote using Memory-based CF for the active user a on item j, $mem\_cf\_PV_{aj}$, is a weighted sum of the votes of similar users:

$$mem\_cf\_PV_{aj} = N \sum_{b \in kNN} sim\,(a,b)\,(v_{bj} - \overline{v}_b) \tag{1}$$

where $\overline{v}_b$ is user b's average vote and weight $sim(a,b)$ represents the correlation or similarity between the active user a and each user b. kNN is the set of k Nearest Neighbours to the current user, as illustrated in figure 6.

A neighbour is a user who has a high similarity value, $sim(a,b)$, with the active user. The set of neighbours is sorted in descending order of similarity; the technique used to measure similarity is described shortly. Experiments in the following chapter use a value of $k = 8$ neighbours. A large number of neighbours can translate into a larger number of items for which recommendation is possible, although, it might be the case that the identified neighbours are not of close similarity which leads to reduced recommendation accuracy. Thus, a trade-off is required and $k = 8$ is a reasonable value. N is the normalising factor such that the absolute values of the weights sum to unity.

From equation 1, a user's vote can be predicted for any item. In the context of this work, the likelihood of a reuse opportunity for any method in the component library can be predicted based on the active developer's current preferences. An assumption is made that every method in the component library has been voted for by at least one user previously. Library methods are ranked based on

Table 2
Vector Space Model (VSM) Item-User Matrix for 5 Users

Component Method	User Vectors				
	U1	**U2**	**U3**	**U4**	**U5**
JL:setBorder(Border)	1	1	0	0	1
JF:setDefaultCloseOp(int)	1	1	0	0	0
JP:add(JLabel)	1	3	1	0	1
JP:setBorder(Border)	1	0	3	4	0
JF:setContentPane(JPanel)	1	0	1	2	0
JF:setSize(int,int)	1	0	1	2	0
JF:setVisible(boolean)	1	1	1	2	0

their predicted vote and the top n methods, with the highest vote, are recommended to the developer. Based on informal observation, the experiments in the next section use a maximum value of $n = 7$. That is, a recommendation set can contain, at most, seven components. Cognitive and software engineering studies have shown that seven is an understandable number of options to present to a developer [39,59].

Comparing Users Central to CF is the ability to determine a set of users who are most relevant or similar to the active user for whom the recommendation is being sought. This corresponds to function $sim(a, b)$, introduced in equation 1. In this research, the *Vector Space Model* [24] is employed. A user vector simply records a user's invocation count (vote) for all items that can be found in the component library. The order of invocations is ignored; ordering information is used later in the Content-Based Filtering algorithm, as discussed in section 5.3. Table 2 illustrates an example of the Vector Space Model. In total, there are 7 library items and 5 user vectors. Each user vector contains an invocation count for all library items. A user corresponds to a Java method as defined in section 5.1. A user's vector is collected implicitly by analysing their source code. The similarity between two vectors, belonging to user a and user b, is calculated as follows:

$$sim\,(a,b) = \sum_{j} \frac{v_{aj}}{\sqrt{\sum_{k \in I_a} v_{ak}^2}} * \frac{v_{bj}}{\sqrt{\sum_{k \in I_b} v_{bk}^2}} \quad (2)$$

where the squared terms in the denominator are used to ensure that a user who uses many components does not appear more similar to other users. Vote v_{aj} and v_{bj} corresponds to each user's vote for item j. I_a and I_b are the sets of items the user a and user b has voted on respectively. The cosine of the angle formed between user a's vector and user b's vector, $sim\,(a, b)$, will fall in the range $[0, 1]$[1]. A cosine of 1 indicates two users are identical, whereas 0 denotes no similarities. More details can be found in [30].

Algorithm Characteristics To determine the kNN nearest users, the active user must be compared with each user in the code repository. This property of Memory-based CF algorithms has potential implications for scalability and runtime performance. In general though, Memory-based CF works reasonably well, is straightforward to implement, and easily allows new users to be added.

5.2.2. Model-based Collaborative Filtering

Model-based CF algorithms seek to fit a probabilistic model to the user database through unsupervised learning techniques and to subsequently use this model to quickly make predictions. From a probabilistic perspective, the task of Model-based CF is to determine the probability that an active user has a particular vote value for a specified item. Breese et al. [2] describe two types of probabilistic models suited to CF: Bayesian clustering and Bayesian network models. In the first model, like-minded users are clustered together into classes. Each item, given the cluster, is treated independently. A recommendation for an active user is based on the cluster of which they are a member. In the second model, a Bayesian network is employed. In this case, items are no longer treated independently, but rather conditional dependencies between items is learned from the data. From our previous work, it was discovered that Bayesian networks perform poorly in our domain [32]. In addition, clustering has been suggested as a natural preprocessing step for CF [54]. This research uses a similar version of the Bayesian clustering Model-based CF algorithm described by Breese et al. [2].

Clustering Users Initially, a set of source code clusters $C = c_1, c_2, ..., c_m$ are created where

[1] Typically in recommender systems, a user can have a negative vote for an item which results in $sim\,(a, b)$ falling in the range of $[-1, 1]$

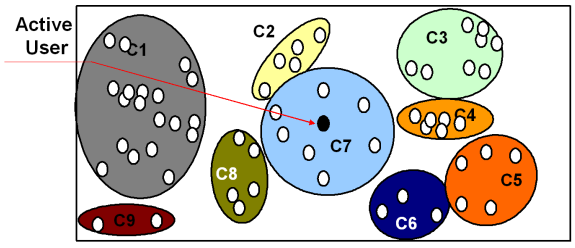


Fig. 7. Illustration of Clustering: The active user must be compared with 9 other clusters as opposed to 43 users when using the *kNN* Memory-based algorithm.

m is the number of unique clusters, as shown in figure 7. The probability that an active user, user a, belongs to a particular cluster c_x, $Pr(c_x, a)$, is estimated. After establishing the cluster of which user a is mostly likely to be a member, a modified version of equation 1 is used to predict the vote for user a on item j:

$$mod\_cf\_PV_{aj} = \overline{v}_a + Pr(c_x, a)(v_{c_x\bar{j}}) \quad (3)$$

where c_x is the classified cluster and $v_{c_x\bar{j}}$ is the average vote for item j in cluster c_x. To produce a recommendation set, the top n items with the highest predicted vote are recommended. Like the Memory-based CF algorithm, a value of $n = 7$ is used in the experiments in the next section.

The clustering algorithm is implemented using a simple Naïve-Bayes model [22]. The probability that user a belongs to cluster c_x is calculated as follows:

$$Pr(c_x|a) = P(c_x)\prod_{j}^{n} P(v_j|c_x) \quad (4)$$

where n is the number of items, j is one library item and $P(c_x)$ is the prior probability of this particular cluster c_x. $P(c_x)$ can be estimated by counting the training examples that fall into cluster c_x and dividing by the size of the training set. $P(v_j|c_x)$ is the probability of the active user's vote for item j given that this user is a member of cluster c_x. By examining all clusters, it can be determined which cluster has the highest posterior probability given the active user. That cluster is then used by the recommendation algorithm in equation 3. Full implementation details of this algorithm can be found in [31].

Characteristics A considerable benefit of Model-based CF is that it only requires the parameters of the model to be kept in memory when making predictions. This requires less space than the entire *item-user* matrix required by the Memory-based approach. Unlike the Memory-based scheme, the Model-based technique requires an initial learning phase and it can be difficult to add new users.

5.3. Content-Based Filtering (CBF) Recommendations

Like Collaborative Filtering, the goal of *Content-Based Filtering* (CBF) [50] is to suggest, or to predict, the utility of certain items for a particular user. CBF recommendations are based solely on an analysis of the items for which the current user has shown preference. Unlike CF, users are assumed to operate independently. Items which correlate closely with the user's preferences are likely to be recommended. For example, in a news recommender system, keywords from the current user's preference would be analysed to recommend news stories which contain similar keywords.

We propose that software components are commonly used, by developers, in similar patterns and that by examining these patterns future component use can be predicted. In this work, the CBF algorithm examines the order in which a user employs a set of methods from a component library. This novel technique is quite unlike standard CBF systems. As stated in subsection 4.2.5, the item invocation order, for each user, is stored in a list. From this, it can be determined that user a invoked item x followed by item y. When making a Content-Based prediction for item j to the active user a, the set of all users who voted for item j needs to be examined. Assuming there is only one user, user b, in the code repository that has voted for item j, then the predicted vote for user a on item j is calculated as follows:

- Firstly, the last item that the active user a invoked, $Prev_a$, is reviewed.
- The item that user b invoked previous to item j, $Prev_{bj}$, is also examined.
- If $Prev_a$ is identical to $Prev_{bj}$, then intuitively, and based on our earlier proposal, it is likely that the active user will want to invoke item j next.

5.3.1. Worked Example

For clarity, the CBF algorithm is explained with an example based on figure 8. When making a recommendation for the active user a on the item `setSize(int,int)`, which is used by a randomly selected user b, a check is done to see if the last item invoked by the active user (`setContentPane(JPanel)`) is the same as the item user b invoked previous to `setSize(int,int)`. These two items are the same, therefore, it is expected that `setSize(int,int)` would be recommended to user a next. Using CF, `setSize(int, int)` and `setVisible(boolean)` would be recommended equally with the same predicted vote, assuming user b is a close neighbour of user a or that user a is a member of the same cluster as user b. The CBF technique ensures `setSize(int,int)` is recommended first but could result in `setVisible(boolean)` not being recommended.

Unlike our previous work that only investigated the sequence of two method invocations [30], this technique is extended to look back for the longest method invocation-order similarities between two users. A long sequence of invocations, identical to the active user's, helps to produce recommendations with a high confidence value. Analysing the item preferences of only one other user, user b, the correlation or similarity between user a and user b's invocation lists, $sim(a, b)$, is defined as follows:

$$sim(a,b) = \frac{|U_{ab}|}{|I_a|} * \frac{index_a}{|I_a|} \tag{5}$$

where I_a is the set of items user a has voted on, U_{ab} is the longest set of items that both user a and b invoked in identical order. The *index* variable in this novel algorithm denotes the most recent position where both users invoked the same item. Consider another example where user a and user b both invoked 10 methods and the method `setSize(int,int)` is invoked twice by each user. The method `setSize(int,int)` is the first method invoked by user a and is subsequently invoked again, by user a, at position 8 (where position 10 would be the last method invoked). Similarly, user b invoked the method `setSize(int,int)` at position 2 and position 6. In this example, $index_a$ would equal 8 and $index_b$, as introduced shortly, would equal 6. Ideally, $index_a$ will equal $|I_a|$, i.e. the last method invoked by the active user.

In figure 8, there are two users: a and b. The size of the set of items that user a invoked, I_a, is 5. The size of the set of items that user a and user b invoked in identical order, U_{ab}, is 4. The $index_a$ is 5 as `setContentPane(JPanel)` is the last identical item that both user a and user b invoked, and hence $sim\,(a, b)$ evaluates to 0.8. The index for user b, $index_b$, is also stored which is needed in equation 6; $index_b$ is 6 in this example. The index value has significance. The aim of this recommendation is to predict the next invocation for user a, and thus order similarities towards the latter part of user a are the most important.

The previous example only considered one user in the code repository that invoked item j. A new recommendation algorithm that considers all users in the code repository is now presented. The predicted vote for user a on item j, cbf_PVaj, is defined as follows:

$$cbf\_PV_{aj} = N \sum_{b \in A} sim\,(a,b)\,(bjNextInvoked) \tag{6}$$

where A is the set of all users in the code repository who voted for item j, N is the normalising factor such that the absolute values of the weights sum to unity, and $sim(a, b)$ is calculated using equation 5. $bjNextInvoked$ is a boolean value that simply denotes whether item j is invoked directly after user b's index ($index_b$). Referring back to figure 8, and assuming user b is the only user in the code repository, if a prediction is being made for user a's vote on the item `setSize(int,int)` then $bjNextInvoked$ would equal 1. If, however, a prediction is being made for user a's vote on the item `setVisible(boolean)` then $bjNextInvoked$ would equal 0, and this method would not be recommended. From equation 6, the vote for any item using CBF can now be predicted.

Characteristics Compared with regular Collaborative Filtering, it would appear that CBF is computationally expensive; for example, the CBF algorithm must find the longest sequence of invocation similarities. Notably though, the number of item invocations for each user is typically quite small; based on the dataset used in this work, we found each user invokes an average of 11 items. Additionally, unlike the CF technique where all users are exam-

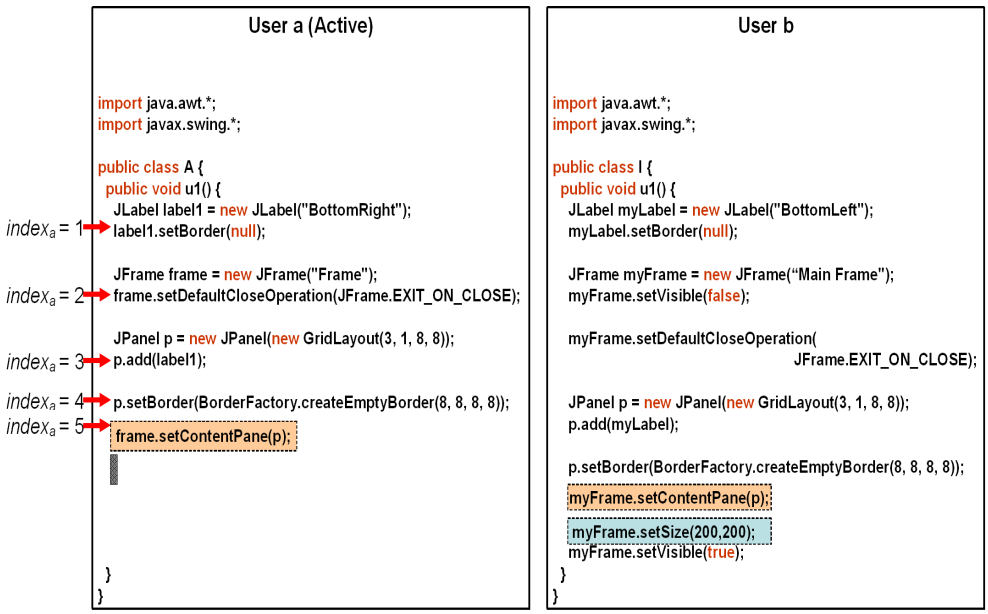


Fig. 8. Using Collaborative Filtering, RASCAL will recommend that the active developer, user a, should invoke JFrame methods `setSize`() and `setVisible`(). However, it is desirable that `setSize`() is invoked before `setVisible`().

ined, a CBF recommendation for an item j only considers user who have invoked that item previously; this is likely to increase performance.

The ability of this CBF algorithm to predict component usage, based on component ordering, is an important advantage over the CF algorithms explained earlier. Though, it could be argued that the CBF recommendations are short-sighted, whereas CF recommendations allow a developer to better anticipate future component use. Additionally, it can be stated that developers do not invoke methods in order; in such circumstances CF algorithms are more appropriate.

When making a recommendation, the Content-Based Filtering algorithm previously described examines several users to determine the order in which they invoke particular methods. Some practitioners may argue that this filtering algorithm does not strictly confirm with the standard definition of CBF, as users are not treated independently. The validity of these concerns is recognised and we acknowledge that our algorithm widely differs from standard CBF techniques. However, it is still believed that the term CBF adequately describes this approach. Currently when the predicted vote of an item is calculated, each user of that item is examined. With little modification, knowledge relating to the invocation order of items could be initially learned by analysing the source code repository. For example, it could be learned that `setX`() is typically invoked prior to `setY`(). Subsequently, this knowledge could be reused in future recommendations, eliminating the need to examine other users. This corresponds with the idea of learning information pertaining to genre in a movie or music CBF recommender system. Additionally, this modification would improve the performance of equation 6.

5.4. Hybrid Recommendations

This subsection describes how the previous two filtering techniques can be merged. By merging these filtering algorithms, it is anticipated that a recommendation set will contain a broad set of useful recommendations in a meaningful order.

5.4.1. Memory-based Collaborative Content-Based Filtering

The similarity measure, $sim(a, b)$, defined in equation 2, and used in the Memory-based CF equation 1, can be extended to include the invocation order similarity defined in equation 5. Each

similarity value is given an arbitrary weighting of $ws = 0.5$, where ws is the predefined weight given to the CF measure. Similarity is now determined as follows:

$$sim\,(a,b) = \left(\frac{|U_{ab}|}{|I_a|} * \frac{index_a}{|I_a|} * (1-ws)\right) + \left(ws * \sum_j \frac{v_{aj}}{\sqrt{\sum_{k\in I_a} v_{ak}^2}} * \frac{v_{bj}}{\sqrt{\sum_{k\in I_b} v_{bk}^2}}\right) \quad (7)$$

The CBF algorithm also needs to be modified. Currently using CBF, a predicted vote for an item is based on every user, in the code repository, of that item. This is updated so that only the nearest neighbours are examined, as determined using equation 7. Equation 6 is modified as follows:

$$mem\_cbf\_PV_{aj} = N \sum_{b\in kNN} sim\,(a,b)\,(bjNextInvoked) \quad (8)$$

where kNN is the set of nearest neighbours. The final equation for calculating the predicted vote for active user a on item j using *Memory-based Collaborative Content-Based Filtering*, $mem\_ccbf\_PV_{aj}$, is as follows:

$$mem\_ccbf\_PV_{aj} = (mem\_cf\_PV_{aj})\,(w) + (mem\_cbf\_PV_{aj})\,(1-w) \quad (9)$$

where w is the predefined weight given to the CF prediction value. Initially, the predicted vote for an item, using standard Memory-based CF, $mem\_cf\_PV_{aj}$, is calculated using equation 1. This examines each neighbour's vote for this item. The predicted vote is then weighted. The result is added to the Content-Based Filtering predicted vote. Equation 8 is used to calculate the CBF vote, $mem\_cbf\_PV_{aj}$. This also examines each neighbour to determine if item j is directly called after the neighbour's $index_b$ variable. The $index$ variable denotes the most recent position where both the active user and its neighbour invoked the same item. The CBF vote is also weighted. The experiments in the next section use a arbitrary value of $w = 0.5$.

5.4.2. Model-based Collaborative Content-Based Filtering

This subsection again presents a new algorithm for producing recommendations. The probability measure, $Pr(c_x, a)$, is the probability that user a belongs to cluster x. Using the probability equation 4, it can be determined which cluster that user a has the highest posterior probability of belonging to. Assuming this is determined, then the CBF algorithm defined in equation 6 can be modified as follows:

$$mod\_cbf\_PV_{aj} = N \sum_{b\in c_x} sim\,(a,b)\,(bjNextInvoked) \quad (10)$$

where c_x is the cluster that user a is most similar too. This cluster is likely to contain several unique users. Using the CBF technique, a vote is only predicted for all the items which the users in cluster c_x have invoked at code-writing time. Our final equation for calculating the predicted vote for the active user a on item j using *Model-based Collaborative Content-Based Filtering*, $mod\_ccbf\_PV_{aj}$, is as follows:

$$mod\_ccbf\_PV_{aj} = (mod\_cf\_PV_{aj})\,(w) + (mod\_cbf\_PV_{aj})\,(1-w) \quad (11)$$

where w is the predefined weight given to the CF prediction value. Initially the predicted vote for an item is calculated using the standard Model-based CF equation 3, $mod\_cf\_PV_{aj}$. This is then weighted. The result is added to the CBF predicted vote. To calculate the CBF vote, $mod\_cbf\_PV_{aj}$, equation 10 is used. The CBF vote is also weighted. The experiments in the next section use a arbitrary value of $w = 0.5$.

5.4.3. Characteristics

Referring back to figure 8, the hybrid Memory-based equation 9 and Model-based equation 11 are designed to ensure both `setSize(int,int)` and `setVisible(boolean)` are recommended. However, `setSize(int,int)` will have a higher predicted vote and will appear before `setVisible(boolean)` in the recommendation set. The next section evaluates these algorithms.

6. Evaluation

6.1. Dataset

Over 35,000 recommendations were produced for 3481 methods mined from Sourceforge [40]. Each method invoked an average of 11 items from the component library. Recommendations were made for both the *SWING* and *AWT* libraries; in total there was 2090 library methods that were invoked at least once in our code repository. Since we have the complete source code, a recommendation for a piece of code can be automatically evaluated by checking whether the recommended method was in fact called subsequently.

For each user method, several recommendations were made. For clarity, an example is given. One user, a fully developed Java method, is taken from the code repository. This user is now the active user and will not be used by any of the filtering algorithms when making a recommendation. In total, this active user calls ten Swing library methods (items) at the code-writing stage, as opposed to runtime. The tenth invocation is automatically removed from the active user's *invocation list* and a recommendation set is produced for the active user based on the preceding nine invocations. The aim is to verify if the proposed recommendation methodology can predict the missing method. Following this recommendation, the ninth invocation is also removed from the active user's invocation list and a new recommendation set is formed based on the preceding eight invocations. This process is continued until all-but-one invocation remains.

6.2. Evaluation

Precision and Recall are the most popular metrics for evaluating information retrieval systems. Precision is defined as the ratio of relevant recommended items to the total number of items recommended; $P = n_{rs}/n_s$, where n_{rs} is the number of relevant items selected and n_s is the number of items selected. This represents the probability that a recommended library method is relevant. A library method is deemed relevant if it is used by the developer for whom the recommendation is being sought. Recall is defined as the ratio of relevant items selected to the total number of relevant items; $R = n_{rs}/n_r$, where n_{rs} is the number of relevant items selected and n_r is the number of relevant items. This represents the probability that a relevant library method will be recommended. Several approaches have been taken to combine precision and recall into a single metric. The $F1$ measure, initially introduced by van Rijsbergen [55], combines both with an equal weight in the following form: $F1 = 2PR/(P + R)$.

It is particularly important that RASCAL recommends methods in a relevant order i.e. the invocation order. We will evaluate this using a simple binary Next Recommended (NR) metric; $NR = 1$ if we successfully predict or recommend the next method a developer will use, otherwise $NR = 0$. The time taken to query a library can strongly impact a developer's perception of that library. A developer will become frustrated if retrieval times are long and thus we also evaluate this.

These automated experiments cannot evaluate the impact of incorrect or irrelevant recommendations. There is also an assumption that the developers of the original source code made optimal use of the available component libraries, and thus relevancy or irrelevancy can be precisely defined. We acknowledge this limitation of our existing work and the need for practical experimentation to fully evaluate RASCAL.

6.3. Results

Figure 9 displays the results for each of the aforementioned algorithms. Result values are similar, and so for clarity, the average result of each technique is listed in table 3. For both the pure and hybrid model-based CF algorithms, the number of cluster is 850. This is a significant reduction from the original 3481 users used in the other algorithms. A baseline result is also presented. Baseline recommendations were produced by recommending, at each recommendation stage, the seven most commonly invoked library methods.

Two universal trends quickly become apparent in the following results: precision typically decreases as more information is known regarding the active user, while recall increases. Consider an active user who uses a total of ten library methods. When a recommendation is made for this user, when only one library method is simulated as being used, then there is a set of nine possible remaining methods to recall. The chances of recalling all relevant methods is quite low and hence the recall result is lower in earlier recommendations.

However, when this user has called 9 methods and there is only 1 possible method left to recall, then the chances of this method being in the recommendation set is quite high. In contrast, the more invocations mimicked as being invoked, the fewer there are to correctly recommend, and hence, there is a greater probability of recommending an irrelevant library method. Thus, precision decreases in latter recommendations. The results of all techniques are compared below.

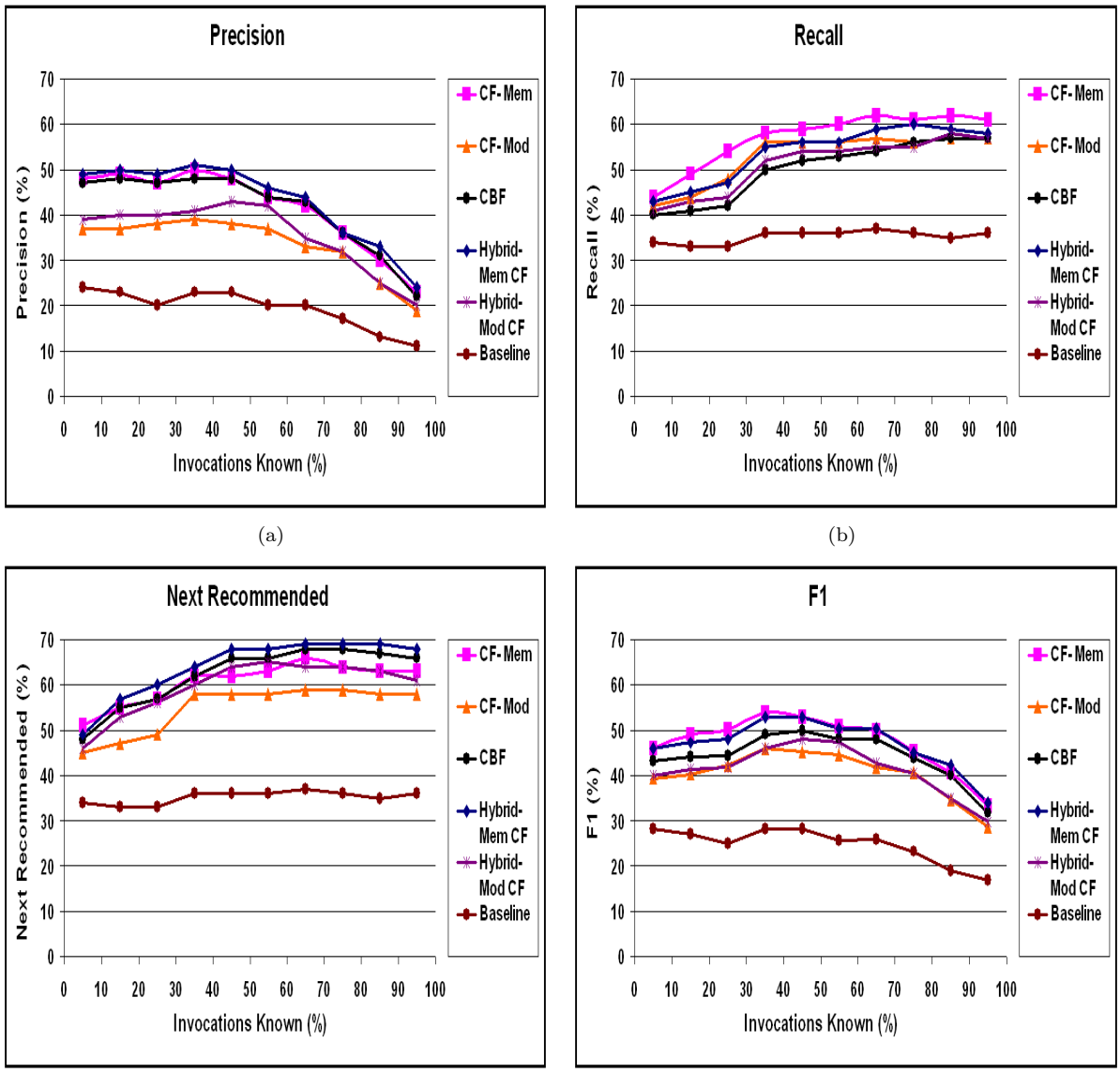
Precision The superior recommendation technique, based on precision, is a hybrid of Memory-based CF and CBF. The average precision is 43.2% when using this algorithm. This compares with 41.8% when using only the Memory-based CF algorithm and 41.4% when using the CBF algorithm. Notably, all three algorithms produce similar results. The hybrid Model-based technique does not perform as well, only achieving a precision value of 35.7%. Though, this is an improvement on the pure Model-based CF precision of 33.5%. The poor performance of the Model-based technique suggests that it is difficult to develop a cluster model that fits the data well. As a result, the accuracy of recommendations suffer. It is believed that members in a cluster agree on many items (based on the recall results that follow) but that there is wide disagreement among cluster members on other items, thus reducing precision.

Recall Memory-based CF, at 57%, produces the best recall value of all algorithms. When either of the CF algorithms is merged with the CBF algorithm, the recall value drops. From a CBF perspective though, merging a CF algorithm with a CBF algorithm improves recall. The hybrid Memory-based algorithm achieves an average recall value of 53.8%, while the Model-based hybrid approach has an average recall value of 51.3%. In terms of supporting software reuse, all algorithms are very effective at recalling relevant components. Pure CF examines a general set of neighbours when making a recommendation, based on all of the current user's invocations. Contrarily, CBF only examines users that have invoked the particular item for which recommendation is sought. Therefore, it is understandable that the CF algorithm produces a broad recommendation set that recalls more relevant components while CBF technique is short sighted and only recalls immediately relevant components.

Next Recommended A hybrid of Memory-based CF and CBF is the most effective technique for predicting the next method that a developer uses. Using this approach, the average NR value is 64.1%. This compares with 60.6% when only the Memory-based CF algorithm is used and with 62.3% when only the CBF algorithm is used. The Model-based CF algorithm significantly improves when merged with the CBF technique. Originally, the average NR value was 54.9%, but this now increases to 59.6%. It is expected that the hybrid algorithm and the pure CBF algorithm perform best as these are the only techniques to consider the order of invocations, and thus the most likely to identify the next useful item.

F1 The pure Memory-based CF algorithm produces the best F1 result, with an average of 47.2%. This is followed closely by the Memory-based hybrid algorithm which has an F1 value of 47%. Again, the efficient Model-based CF algorithm improves when merged with the CBF technique. The F1 value increases from 40.4% to 41.2% when using the latter hybrid approach.

Time Memory-based CF recommendations take an average of 1.2 seconds, the Model-based CF algorithm usually produces a recommendation within 0.2 seconds, and CBF recommendations typically take 0.72 seconds. The hybrid Memory-based Collaborative Content-Based Filtering algorithm produces recommendations, on average, in 1.5 seconds. The Model-based Collaborative Content-Based Filtering algorithm takes an average of 0.31 seconds to produce a recommendation. With the Memory-based hybrid approach, each user in the code repository needs to be examined to determine the active user's set of nearest neighbours. Additionally, a CBF similarity measure must also be calculated for all users, this decreases recommendation efficiency. However, using the Model-based hybrid technique, only each user in the selected cluster needs to be examined to determine CBF similarity. Therefore, the efficiency of the Model-based CF algorithm is not adversely effected when it is merged with CBF.



(a) (b)

(c) (d)

Fig. 9. Recommendation Results: (a) Precision (b) Recall (c) Next Recommended (d) F1

Table 3
Results Summary: Average Precision, Recall, Next Recommended, F1, and Time

Filter	Pr	Rc	NR	F1	Time	Comment
CF: Memory-based VSM	41.8%	**57.0%**	60.6%	**47.2%**	1.20s	Great results in reasonable time.
CF: Model-based Naïve-Bayes	33.5%	52.9%	54.9%	40.4%	**0.20s**	Reasonable results in excellent time.
CBF	41.4%	50.2%	62.3%	44.2%	0.72s	Great NR result.
Hybrid: Memory CF and CBF	**43.2%**	53.8%	**64.1%**	47.0%	1.50s	Great results in reasonable time.
Hybrid: Model CF and CBF	35.7%	51.3%	59.6%	41.2%	0.31s	Reasonable results in excellent time.
Baseline	19.4%	35.2%	33.7%	24.68%	0.09s	Poor Results.

6.4. Results Conclusions

If a relatively small dataset is used, as in these experiments, then the hybrid Memory-based Collaborative Content-Based Filtering algorithm is ideal. Recommendations are of a good quality and can be produced within a reasonable time. Out previous research, when compared with these results, has shown that increasing the size of the code repository does not actually improve recommendations [33]. Rather, this can be achieved by populating the code repository with users who frequently use the component library. For example, in these experiments, each user had a minimum of eight library method invocations.

If the number of library components increases dramatically, then it may be necessary to add many new users to the code repository. In such a situation, the hybrid Memory-based algorithm becomes inefficient, as each user in the code repository must be examined. Instead, the Model-based hybrid technique should be used, as this can produce good quality recommendations efficiently. This efficiency is due to the number of clusters not growing linearly with the number of users. In comparison with the Memory-based hybrid algorithm, this technique results in a small decrease in recommendation accuracy.

The results presented here are heartening. Significant values of precision, recall, NR, and F1 can all be achieved. Further, recommendations can be produced very quickly. As a means of supporting component reuse, these recommendation algorithms prove highly effective. Average recall, for all algorithms is over 50% while the average NR result is as high as 64% when using the hybrid memory algorithm. An interesting future direction would be to compare the success of RASCAL across different domains; for example, current experiments are based on the Swing and AWT Graphic User Interface (GUI) libraries. It is possible that such libraries encourage a linear style of programming and thus improve RASCALs performance. Further analysis is merited

Based on the current results, a reuse support tool, such as RASCAL, will greatly assist a developer. In particular, RASCAL is capable of identifying reuse opportunities that may not be obvious to the developer. Depending on an individual's environment or circumstances, the importance of each of the evaluation criteria described earlier will differ. Therefore, the RASCAL tool can be configured to use either of the five variations of the recommendation algorithm.

7. Related Work

This section reviews related work in the area of component retrieval and software reuse support. For background purposes, traditional retrieval techniques are detailed. These rely heavily on primitive IR and descriptive techniques. Following this, more sophisticated retrieval and recommendation techniques are presented that typically employ intelligent IR approaches.

7.1. Traditional Component Retrieval

Traditional retrieval schemes focused generally on basic IR techniques [9], and on descriptive methods such as *keyword search* [29] and *faceted classification* [44]. Other syntactic-based approaches such as *signature matching* have been used by Zaremski and Wing [60] for retrieval. Many of these techniques are not ideal. When using a keyword search, too many or too few components may be returned because only keywords are used in the search. Signature matching queries are strict and formal. Additionally, signature alone does not guarantee the expected behaviour of the component. With faceted classification, the taxonomy must be well defined and each component must explicitly fit the classification scheme.

Many of these techniques have been extended to include *semantic-based* retrieval features. Typically while querying the component library, the developer specifies component requirements using natural languages which are then interpreted using a language ontology as a knowledge-base. Components in the repository also have a natural language description. Both the developer query and component descriptions are formalised and closeness is computed. A set of potentially reusable components are then ranked based on their closeness value. Unlike the earlier retrieval schemes, domain information and component relationships are generally considered. Empirical results indicate that semantic-based retrieval schemes are superior to traditional approaches [52].

7.2. Intelligent Component Retrieval

More recently, several intelligent component retrieval tools have been proposed. Gu et al. [13] present a *Conversational Case-based component Retrieval Model* (CCRM) that helps developers locate reusable components. The previous retrieval techniques assume that a developer can clearly and accurately formulate queries. In this approach, software components are represented as cases. A knowledge-intensive case-based reasoning method is adopted to explore context-based semantic similarities, between a developer's query and stored components. A conversational approach is used to collect developer requirements interactively and incrementally. Like our work, the authors recognise that developers find it difficult to express their reuse intentions and use discriminative questions to extract further information from a developer and to more clearly define the query. This tool is useful but lacks the ability to proactively infer a developer's reuse requirements.

Just as we employ a software agent to deliver component recommendations to a developer, a software agent has been developed by Drummond et al. to assist library browsing [6]. The authors describe active browsing: an active browser suggests to the developer components it estimates to be close to the target search. A developer uses the library browsing system for retrieving components in a normal manner. In addition, the active agent attempts to recognise the developer's intent and subsequently provides guidance that accelerates the search. Based on experimental results, the agent identified the developer's search goal 40% of the time before the developer reached the goal. This work is intended to complement, rather than replace, browsing.

7.3. Web-based Retrieval

Several web-based commercial and open-source component retrieval techniques have been developed such as *ComponentSource* [3], and *Planet-Sourcecode* [42]. Similar to the *Google* search engine [41], *ComponentRank* [17] is one such technique that harnesses web-based search approaches. Components are ranked based on analysing *use relations* among the components and propagating the significance of a component through the use relations; this is similar to our technique which examines how components are used in practice. This technique is effective at giving a high rank to stable general components, which are likely to be highly reusable, and a lower rank to non-standard specialised components.

Hummel and Atkinson [16] have carried out a study on using the web as a reuse repository. They evaluated several specialised source code search engines such as *PlanetSourceCode* and *Koders* [20], and more general search engines such as *Google* and *Yahoo*. They discovered that there is a magnitude of accessible code resources on the web. Many of these specialised search engines offer features for choosing language, selecting syntactic requirements such as a method name or parameter type, and accessing API information. However, they discovered that the non-specialised search engines produced comparable results. They also identified some of the advantages of web-based approaches such as scalability and efficiency, and noted limitations such as security and legal concerns. Such resources could potentially be used by RASCAL to learn new component information.

7.4. Component Retrieval by Example

Reuse support through code examples is a popular research theme. Grabert and Bridge [12] present a software tool for *examplet reuse*. They define examplets as consisting of two parts. The first is a snippet of source code, Java in their example, which shows how to accomplish a task using library components. The second part of the examplet is a statement of the source code goal in free text. They have developed a tool that allows developers to specify their reuse goal in natural text. Like RASCAL, this tool automatically takes into account the source code that the developer is working on. The system combines text retrieval with a semantic net representation of the source code to retrieve relevant examplets. While it is feasible that example source code is available which uses library components, it is less likely that all source code will have an associated examplet description.

Another notable tool for finding code examples is *Strathcona* [14]. This tool is used to find relevant source code in an example repository of many projects by matching the code a developer is currently writing with all code in the example repository. Similarity is based on multiple structural

matching heuristics, such as examining inheritance relationships, method calls, and class instantiations. These measures are applied to the code currently being written by the developer and, subsequently, matched examples from the repository are retrieved and recommended to the developer. Unlike the work of Grabert and Bridge [12], queries are automatically created by *Strathcona* once a developer requests an example piece of code. Like our work, this tool benefits from being able to populate its example code repository from any code or framework irrespective of coding conventions, as all code incorporates structure.

7.5. Proactive Recommendation: CodeBroker

A common shortcoming of the above solutions is that they are developer dependent. A developer must be capable of anticipating all reuse opportunities and initiating the retrieval process. However, if a developer believes a reusable component for a particular task does not exist then they are less likely to search the component repository. Thus, *no-attempt-to-reuse* is made.

Ye et al. [57,58] present a proactive recommender tool named *CodeBroker* that addresses this key problem. *CodeBroker* infers a developer's need, or opportunity, to reuse a set of software components, and then proactively recommends these. The need for a component is inferred by monitoring the comments a developer writes along with method signatures. Based on this, a set of matching components is retrieved and presented to the developer for reuse. This solution greatly improves on previous approaches. Nevertheless, the technique used to implement this approach is not ideal. Reusable components in the component library must be sufficiently commented to allow matching; this may, in effect, exclude the possible recommendation of many existing components. In addition, developers must also actively and correctly comment their code. These pragmatic concerns restrict the applicability of *CodeBroker* in an industrial setting.

This work is strongly related to *CodeBroker*, and to a lesser extent with Strathcona [14]. Like Rascal, *CodeBroker* recommends a set of methods from the component library that a developer should consider invoking. This differs from the Strathcona tool which delivers a set of code examples to a developer. At present, our experiments are automated and reasonably large, use source code from a variety of open-source projects, and recommend methods from the Swing and AWT *GUI* component libraries. Contrarily, *CodeBroker* is evaluated manually using five programmers and a small number of programming tasks, requires HTML-based online documentation that is generated by running Javadoc over Java source programs, and produces recommendations for the Java 1.1.8 Core API library and Java General Library (JGL) [56]. As *CodeBroker* is a fully developed tool, previous *CodeBroker* experiments tended to focus on the cognitive aspects of the research whereas in this work the focus is largely on the accuracy of the recommendation algorithms. This makes a direct comparison difficult; nonetheless, it is believed that our technique is the easiest to adopt and the most effective recommender in terms of correct ordering, while *CodeBroker* is a more mature system having been evaluated in practical setting.

7.6. Review

Modern research on component retrieval tends to take a more comprehensive approach than its traditional counterparts. Gu et al. [13] and Drummond et al. [6] focus on developers. They advocate the need for an iterative approach to query formulation as it is often difficult for a developer to initially express their reuse intentions clearly. Inoue et al. [17] consider component quality. They assess how components are used in practice, and rank components based on this. Web-based retrieval techniques focus on scalability, performance, and availability. Component retrieval by example techniques [12,14] acknowledge the importance of program context and the development environment. Queries using this approach consider the source code that the developer is currently working on. The *CodeBroker* tool [58] recognises developer cognitive challenges and aims to address these through proactive component recommendation.

Our work is similar to a number of the areas mentioned above but several key differences exist that distinguish this work from others. We have presented an autonomous software agent that can automatically identify reuse opportunities and, based on this, recommends a set of relevant reusable components to an active developer.

Unlike existing component retrieval tools, the developer does not need to initiate retrieval or perform any additional tasks. The RASCAL agent learns knowledge from existing source repositories and uses this knowledge to automatically identify reuse opportunities.

Component recommendation techniques vastly improve on passive component retrieval tools. Tools, such as *CodeBroker* and *Strathcona*, have important advantages over our work. For example, these tools are fully developed, provide source code examples that assist developers to understand the rationale behind recommendations, and have been evaluated in user trials. Similarly, RASCAL has several distinct advantages. Our tool places a strong emphasis on the order of component recommendations and this is evaluated using the *Next Recommended* metric. In addition, our tool is not dependent on any particular coding styles or conventions, does not record developer coding histories, and does not place additional requirements on developers. This, combined with our efficient algorithms, ensures recommendations are performed quickly. Incorporated within the popular *Eclipse* IDE, RASCAL is an effective, efficient, and easy-to-use component support tool.

8. Conclusion

Software reuse and Component-Based Development is difficult, hampered by the inadequacy of existing component reuse support strategies and tools. Key concerns are the inability of support tools to automatically identify reuse opportunities and formulate reuse queries; the separation of reuse from development tasks; and the lack of techniques to store and distribute task-relevant component knowledge among developers. We have argued that knowledge collaboration tools can be employed to effectively support Component-Based Development by identifying unanticipated reuse opportunities. Such tools capitalise on the similarities that frequently exist between programming solutions; through the mining of source code repositories, knowledge can be extracted pertaining to the use of software components and a collaborative knowledge-base can be established. This is then be used to produce component recommendations.

From extensive experiments, we have concluded that our filtering algorithms can efficiently produce reliable recommendations. Each proposed algorithm significantly outperformed the baseline experiment, and each has unique characteristics that will determine their suitability based on a developer's priorities. For instance, a recommendation set, produced using a *hybrid* of *Memory-based Collaborative Filtering* and *Content-Based Filtering*, can recall, on average, 54% of the reusable components that a developer is interested in, while there is a 64% likelihood that the set will contain the next component a developer invokes.

Sophisticated knowledge-based component retrieval and recommendation schemes are no silver bullets, and these alone will not elevate component reuse to a standard software engineering practice. Other non-technical impediments need to be addressed, for example, reuse needs to be nurtured and developers need to be rewarded for their reuse efforts. Nonetheless, this paper has described an intelligent agent solution that will encourage component reuse by utilising knowledge collaboration and component recommendation techniques. This developer-centric methodology is both lightweight and easily adopted. In addition, the support tool described in this research is an important step towards systematic component reuse.

9. Acknowledgements

Funding for this research was provided by the Irish Research Council for Science, Engineering and Technology (IRCSET) under grant RCS/2003/127.

References

[1] Jeffrey M. Bradshaw. *Software Agents*. AAAI Press, 1997.

[2] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *14th Annual Conference on Uncertainty in Artificial Intelligence*, pages 43–52, Madison, Wisconsin, USA, 1998.

[3] ComponentSource. `www.componentsource.com`. 2007.

[4] Josenildo Costa da Silva, Matthias Klusch, Stefano Lodi, and Gianluca Moro. Privacy-preserving agent-based distributed data clustering. *Web Intelli. and Agent Sys.*, 4(2):221–238, 2006.

[5] M. Dahm. Byte code engineering with the bcel api. *Technical Report B-17-98*, 2001.

[6] Christopher G. Drummond, Dan Ionescu, and Robert C. Holte. A learning agent that assists the browsing of software libraries. *IEEE Transactions on Software Engineering*, 26(12):1179–1196, 2000.

[7] R. Due. The economics of component based development. *Information Systems Management*, 17(1):92–95, 2000.

[8] Liesbeth Dusink and Jan van Katwijk. Reuse dimensions. In *Symposium on Software Reusability*, pages 137–149, Seattle, Washington, United States, 1995. ACM Press.

[9] W B Frakes and B A Nejmeh. Software reuse through information retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

[10] William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–ff., 1995.

[11] William B. Frakes and Christopher J. Fox. Quality improvement using a software reuse failure modes model. *IEEE Transactions on Software Engineering*, 22(4):274–279, 1996.

[12] Markus Grabert and Derek Bridge. Case-based reuse of software examplets. *Journal of Universal Computer Science*, 9:627–640, 2003.

[13] Mingyang Gu, Agnar Aamodt, and Xin Tong. Component retrieval using knowledge-intensive conversational cbr. In *19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 27–30, Annecy, France, 2006.

[14] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *27th International Conference on Software Engineering*, pages 117–125, St. Louis, MO, USA, 2005.

[15] Grace Murray Hopper. The education of a computer. In *1952 ACM National Meeting*, pages 243–249. ACM Press, 1952.

[16] Oliver Hummel and Colin Atkinson. Using the web as a reuse repository. In *9th International Conference on Software Reuse*, pages 298–311, Turin, Italy, 2006.

[17] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Shinji Kusumoto, and Makoto Matsushita. Ranking significance of software components based on use relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[18] Stan Jarzabek and Riri Huang. The case for user-centered case tools. *Communications of the ACM*, 41(8):93–99, 1998.

[19] Tetsuro Kakeshita and Miyuki Murata. Specification-based component retrieval by means of examples. In *Proceedings of the 1999 International Symposium on Database Applications in Non-Traditional Environments*, page 411, Washington, DC, USA, 1999. IEEE Computer Society.

[20] Koders. `www.koders.com`. 2007.

[21] Charles W. Krueger. Software reuse. *ACM Computing Survey*, 24(2):131–183, 1992.

[22] Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of bayesian classifiers. In *10th National Conference on Artificial Intelligence*, pages 223–228, San Jose, California, USA, 1992. AAAI Press.

[23] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *28th International Conference on Software Engineering*, pages 492–501, Shanghai, China, 2006. ACM Press.

[24] Todd A. Letsche and Michael W. Berry. Large-scale information retrieval with latent semantic indexing. *Information Science*, 100(1-4):105–137, 1997.

[25] Jingyue Li, Marco Torchiano, Reidar Conradi, Odd Petter N. Slyngstad, and Christian Bunse. A state-of-the-practice survey of off-the-shelf component-based development processes. In *9th International Conference on Software Reuse ICSR*, pages 16–28, Torino, Italy, 2006. Springer-Verlag.

[26] Wayne C. Lim. Strategy-driven reuse: Bringing reuse from the engineering department to the executive boardroom. *Annals of Software Engineering*, 5:85–103, 1998.

[27] Luqi and Jiang Guo. Toward automated retrieval for a software component repository. In *IEEE Conference and Workshop on Engineering of Computer-based Systems*, pages 99–105, USA, 1999.

[28] Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[29] Y. Matsumoto. A software factory: An overall approach to software production. In *ITT Workshop on Reusability in Programming*, Newport, USA, 1993. IEEE Computer Society Press.

[30] Frank McCarey, Mel Ó Cinnéide, and Nicholas Kuskmerick. Rascal: A recommender agent for agile reuse. *Artificial Intelligence Review*, 24(3-4):253–276, November 2005.

[31] Frank McCarey, Mel Ó Cinnéide, and Nicholas Kuskmerick. A recommender agent for software libraries: An evaluation of memory-based and model-based collaborative filtering. In *International Conference on Intelligent Agent Technology*, Honk Kong, December 2006.

[32] Frank McCarey, Mel Ó Cinnéide, and Nicholas Kuskmerick. Recommending library methods: An evaluation of bayesian network classifiers. In *2nd International Workshop on Supporting Knowledge Collaboration in Software Development*, Tokyo, Japan, September 2006.

[33] Frank McCarey, Mel Ó Cinnéide, and Nicholas Kuskmerick. Recommending library methods: An evaluation of the vector space model (vsm) and latent semantic indexing (lsi). In *8th International Conference on Software Reuse*, Torino, Italy, June 2006.

[34] M. D. McIlroy. Mass produced software components. In *NATO Software Engineering Conference*, volume 1, pages 138–150, Germany, October 1968.

[35] Bertrand Meyer. *Object-Oriented Software Construction (2nd ed.)*. Prentice-Hall, Inc., NJ, USA, 1997.

[36] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *Software Engineering*, 21(6):528–562, 1995.

[37] Hefedh Mili, Ali Mili, Sherif Yacoub, and Edward Addy. *Reuse-based Software Engineering: Techniques, Organization, and Controls*. Wiley-Interscience, New York, USA, 2001.

[38] Rym Mili, Ali Mili, and Roland T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445–460, 1997.

[39] George A. Miller. The magical number seven, plus or minus two. *The Psychological Review*, 63:81–97, 1956.

[40] OSTG. Sourceforge.net is owned by the open source technology group inc (ostg), a subsidiary of va software corporation. `http://sourceforge.net`. 2004.

[41] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Technical Report SIDL-WP-1999-0120*, 1998.

[42] PlanetSourceCode. `www.planetsourcecode.com`. '07.

[43] Jeffrey Poulin. The business case for software reuse: Reuse metrics, economic models, organizational issues, and case studies. In *9th International Conference on Software Reuse*, page 439, Italy, 2006. Springer.

[44] Ruben Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991.

[45] R. Rada. *Software Reuse: Principles, Methodologies and Practices*. Ablex Publishing, NJ, USA, 1995.

[46] P. Resnick, N. Iacovou, M. Suchak, P. Bergstorm, and J. Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *ACM Conference on Computer Supported Cooperative Work*, pages 175–186, Chapel Hill, North Carolina, 1994.

[47] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[48] Badrul M. Sarwar, George Karypis, Joseph A. Konstan, and John Reidl. Item-based collaborative filtering recommendation algorithms. In *World Wide Web*, pages 285–295, Hong Kong, 2001.

[49] Arun Sen. The role of opportunism in the software design reuse process. *IEEE Transactions on Software Engineering*, 23(7):418–436, 1997.

[50] Upendra Shardanand and Patti Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, volume 1, pages 210–217, 1995.

[51] David Sprott. Enterprise resource planning: Componentizing the enterprise application packages. *Communications of the ACM*, 43(4):63–69, 2000.

[52] Vijayan Sugumaran and Veda C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.

[53] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, New York, USA, 2002.

[54] L. Ungar and D. Foster. Clustering methods for collaborative filtering. In *Workshop on Recommendation Systems at the 15th National Conference on Artificial Intelligence*, Madison, Wisconsin, USA, 1998. AAAI Press, Menlo Park California.

[55] C.J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[56] Yunwen Ye. *Supporting Component-Based Software Development with Active Component Repository Systems*. PhD thesis, University of Colorado, 2001.

[57] Yunwen Ye and Gerhard Fischer. Information delivery in support of learning reusable software components on demand. In *7th International Conference on Intelligent User Interfaces*, pages 159–166, San Francisco, California, USA, 2002. ACM Press.

[58] Yunwen Ye and Gerhard Fischer. Reuse-conducive development environments. *Automated Software Engineering*, 12(2):199–235, 2005.

[59] Edward Yourdon. *Modern Structured Analysis*. Yourdon Press, Upper Saddle River, NJ, USA, 1989.

[60] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A key to reuse. In *1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 182–190, US, 1993. ACM Press.

[61] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.