Artificial Intelligence Review (2005) 24:253–276 © Springer 2005
DOI 10.1007/s10462-005-9012-8

Rascal: A Recommender Agent for Agile Reuse

FRANK MCCAREY*, MEL Ó CINNÉIDE & NICHOLAS KUSHMERICK
*School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland (*author for correspondence, e-mail: {frank.mccarey, mel.ocinneide, nick}@ucd.ie)*

Abstract. As software organisations mature, their repositories of reusable software components from previous projects will also grow considerably. Remaining conversant with all components in such a repository presents a significant challenge to developers. Indeed the retrieval of a particular component in this large search space may prove problematic. Further to this, the reuse of components developed in an Agile environment is likely to be hampered by the existence of little or no support materials. We propose to infer the need for a component and proactively recommend that component to the developer using a technique which is consistent with the principles of Agile methodologies. Our RASCAL recommender agent tracks usage histories of a group of developers to recommend to an individual developer components that are expected to be needed by that developer. Unlike many traditional recommender systems, we may recommend items that the developer has actually employed previously. We introduce a content-based filtering technique for ordering the set of recommended software components and present a comparative analysis of applying this technique to a number of collaborative filtering algorithms. We also investigate the relationship between the number of usage histories collected and recommendation accuracy. Our overall results indicate that RASCAL is a very promising tool for allowing developers discover reusable components at no additional cost.

Keywords: Agile processes, Agile reuse, recommender agent, content-based filtering, collaborative filtering, software reuse

1. Introduction

Reuse of software components has been proven to be an effective means of reducing development time and costs whilst benefiting the overall quality of the software (Hooper and Chester, 1991; Yongbeom and Stohr 1998). A mature software development organisation is likely to possess a large, growing repository of components from previous projects. As this repository increases in size, so too does the challenge for developers to remain conversant with all components. Various component retrieval techniques have been developed to allow a developer to discover or locate components in which they are interested

(Inoue et al., 2003; Sugumaran and Storey, 2003). In our work, we shift the attention from component retrieval to component recommendation. RASCAL, a recommender agent for software components in an Agile environment, has been developed for two purposes. First we wish to recommend software components that the developer is interested in. Second, and more importantly, we wish to recommend useful components which the developer may not be familiar with or aware of.

Developers are not always eager to use reusable components even if these components may be useful and improve productivity. A *productivity paradox* (Carrol and Rosson, 1987) exists. Although reusable components for solving problems are available, most developers are not motivated to learn these reusable components. Yunwen and Fischer (2002) discuss the reasons behind the lack of motivation. A developer may give preference to a suboptimal solution as they perceive the time and effort to locate and learn components to be too costly. In addition, the current trend of Agile development advocates minimal support documentation for such components and frequent system releases. Little support documentation coupled with time constraints makes reuse particularly challenging for Agile developers. Agile development and the challenges surrounding reuse in an Agile context is described in detail in Section 3. Several other factors have been documented as to why reuse is often problematic (Frakes and Terry, 1996; Schmidt, 1999) These challenges facing developers are the main motivation for our work. We must assist and encourage developers in making full use of large component repositories by complementing component retrieval with component recommendation. Our goal is to recommend useful components to a developer in a way which is consistent with the principles of Agile development; reusable components currently being developed should not need any additional documentation and reuse of such components should be appealing, straightforward and require little additional effort from the developer.

The RASCAL implicitly gathers information about developer usage histories of components and utilises this information to deduce or infer the need for a particular component or set of components. Specifically the components referred to are Java methods. These inferred methods are then recommended to the developer. The RASCAL continuously runs in the background, monitoring/updating a developer's usage history and frequently makes recommendations as illustrated in Figure 1. Recommendations are produced primarily using a collaborative filtering approach, however we make use of content-based filtering to order recommendations. A basic principle of collaborative systems

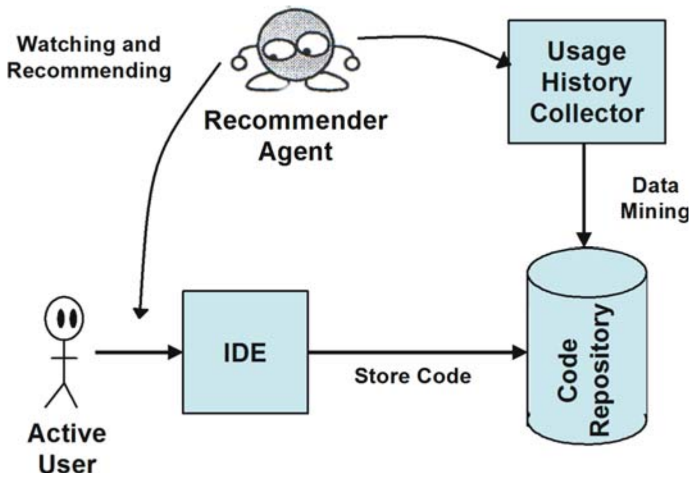


Figure 1. System overview.

is that users can be clustered into groups. Users in a group share preferences and dislikes for particular items and are likely to agree on future items. A recommendation for a user is based on the opinions or ratings of other like-minded users. In Section 5, we demonstrate how this clustering principle can be extended to software classes.

This paper describes the challenges faced by developers when implementing a reuse program in an Agile environment. We also detail an agent for recommending software methods which allows developers discover reusable methods for the purpose of improving developer productivity/quality and promoting software reuse. Repositories of open-source Java code, available from *SourceForge* (OSTG, 2004), are data mined and usage histories of components are automatically collected and stored in a developer-preference database. Based on a number of collaborative filtering approaches, with an extended content-based filtering ordering algorithm, we use the collected component usage histories (user profiles) to recommend to a developer a set of candidate Java methods that may be useful. A comparative analysis of the results from each approach is given along with an investigation into the relationship between the number of usage histories collected and recommendation accuracy.

Several obstacles to reuse have been mentioned. In the following section, we review related work in the area of component search, retrieval and recommendation. In Section 3, we introduce Agile Reuse; we discuss the likely benefits of such an approach and identify the likely implementation difficulties. A brief overview of our system can be found in Section 4 followed by a description of the extended collaborative filtering technique used to produce recommendation sets. Section 6 details experiments and results. Finally in Section 7, we discuss how RASCAL can be extended and draw general conclusions.

2. Related Work

The development of reusable components and component libraries has been an active research area for some time but this alone will not encourage reuse. "A classified collection is not useful if it does not provide a search-and-retrieval mechanism to use it" (Prieto-Diaz and Freeman, 1987). Mili et al. (1998) classify traditional search and retrieval methodologies into four categories, namely *Keyword Search*, *Faceted Classification*, *Signature Matching* and *Behavioural Matching*. Each of these retrieval schemes has a number of limitations that result in less than adequate retrievals. The last two schemes, for example, have been found to be cumbersome and inefficient (Sugumaran and Storey, 2003). A common shortcoming of all these schemes is the failure to take into account the developer or relevant domain information when querying the component repository.

More recently, several *Semantic-Based* retrieval schemes have been proposed; typically while querying the repository the developer specifies component requirements using natural languages which are interpreted using a language ontology as a knowledge base. Components in the repository will also have a natural language description. Both the developer query and component descriptions are formalised and closeness is computed. A set of candidate components can be ranked based on their closeness value. Unlike the approaches mentioned above, domain information, developer context and component relationships are all considered. Empirical results indicate that such schemes are superior to traditional approaches (Girardi and Ibrahim, 1995; Sugumaran and Storey, 2003; Yao and Etzkorn, 2004).

Sourceforge (OSTG, 2004) is a large software repository where various development projects take place using an open-source development model (Raymond, 2004). Developers can search this site to find real development projects, but no support is provided for the retrieval of program components, portions of code, algorithms and many others items which could potentially be reused. *ComponentRank* (Inoue et al., 2003) is a promising component retrieval technique which addresses this issue and is useful for locating reusable components. Similar to *Google* (Page et al., 1998), this approach ranks components based on analysing use relations among the components and propagating the significance of a component through the use relations. Preliminary results indicate that this technique is effective in giving a high rank to stable general components which are likely to be highly reusable and a lower rank to non-standard specialised components.

Drummond et al. (2000) present the use of a *learning* software agent to support the browsing of software libraries. The active agent attempts to learn the component the developer is looking for by monitoring the developer's normal browsing actions. Based on experimental results, 40% of the time the agent identified the developer's search goal before the developer reached the goal. By providing non-intrusive advice that accelerates the search, this work is intended to complement rather than replace browsing.

A major disadvantage with all of the retrieval techniques above is that the developer must initiate the search process. However, in reality developers are not aware of all available components. If they believe a reusable component for a particular task does not exist then they are unlikely to search the component repository; none of the above schemes attempts to address this important issue. Thus to effectively and realistically support component reuse it is tremendously important that component retrieval be complemented with component delivery/recommendation.

CodeBroker (Yunwen and Fischer, 2002) infers the need for components and proactively recommends components, with examples, that match the inferred needs. The need for a component is inferred by monitoring developer activities, in particular developer comments and method signature. This solution greatly improves on previous approaches however the technique is not ideal. The reusable components in the repository must be sufficiently commented to allow matching. Developers must actively and correctly comment their code which currently they may not do. Active commenting is an additional strain placed on developers which may make the use of *CodeBroker* less appealing and particularly unsuitable for Agile development where emphasis is placed on understandable code as opposed to code commenting. A detailed explanation of Agile development is provided in Section 3.

Ohsugi et al. (2002) propose a system to allow users discover useful functions at a low-cost in application software such as MS Word and MS Excel for the purpose of improving the user's productivity;

Convert Text to Table or *Insert Picture* are examples of MS Word functions. A set of candidate functions is recommended to the individual, based on the opinions of like-minded users. The technique used is an extension of traditional collaborative filtering algorithms used in mainstream recommender systems such as *Amazon* (Bezos, 2004) and *Movielens* (2004).

Our work is similar to a number of the techniques mentioned above. We apply the Ohsugi et al. (2002) proposal to a different problem domain, namely reusable software components. Similar to *CodeBroker* (Yunwen and Fischer, 2002) our goal is to recommend a set of candidate software components to a developer; however our recommendations are based on the opinions of like-minded developers and not the developer's comments/method signature. Like Drummond et al. (2000) we use an active agent to monitor the current developer though we are concerned with proactively recommending suitable reusable components as opposed to assisting the search process. Unlike any of the related works our technique is specifically designed to assist reuse in an Agile environment which can be particularly difficult as detailed in the following section.

3. Agile Reuse

Current trends place pressure on organisations to produce new or enhanced software implementations quickly in response to an ever changing environment (Cusumano and Yoffe, 1999). Agile or lightweight software development methodologies were developed primarily to address this problem of producing software in "Internet time" (Turk et al., 2002). Agile methodologies promote early and quick production of working code and allow for changing user requirements. This is achieved by structuring the development process into iterations where the aim of each iteration is to produce working code and other artifacts that provide value to the customer. In many cases source code is the only deliverable that truly matters; the role of analysis, design models, and documentation in the creation and evolution of the software is often marginalised (Turk et al., 2002). Agile principles, found in the Agile Manifesto (Beck et al., 2000), differ greatly from the more traditional rigorous or heavy process models where early up-front design is advocated combined with a strong emphasis on support documentation. Due to their simplistic non-bureaucratic nature and the emphasis placed upon people versus process, Agile methodologies are proving favorable among developers and customers alike, with Extreme Programming (XP) (Beck, 2000) perhaps the best known and widely used Agile methodology (Theunissen et al., 2003).

Software reuse refers to the use of existing artifacts from previous projects as part of a new development project. *Ad hoc* reuse has always existed. However as enterprises invest in developing and maintaining large software systems in an increasingly competitive environment, there exists the need for an effective and structured reuse strategy (Sindre et al., 1993). Ten reusable aspects of any given software project are presented by Frakes and Terry (1996) in their discussion of reuse metrics and models, these include requirements and design reuse. In keeping with Agile principles, we are only concerned with *source code* reuse in our present work. Successful reuse has been shown to improve software quality and developer productivity while reducing overall costs (Hooper and Chester, 1991). A reduction in time to market can also be achieved as detailed by Yongbeom and Stohr (1998); successful industrial examples include (Griss, 1991; Tirso, 1991).

Despite these desirable advantages several factors hamper reuse as discussed in the introductory section. Factors vary from technical difficulties such as support environments to developer attitudes. As reuse becomes more prominent and accepted in industry, systems and tools that aid and support reuse become a key aspect in achieving successful reuse of software artifacts (Daudjee and Toptsis, 1994) This notion is reflected by the shift in software reuse research from initially focusing on techniques to develop reusable components and component libraries to a focus on supporting reuse through intelligent storage and retrieval strategies (Inoue et al., 2003; Sugumaran and Storey, 2003; Yao and Etzkorn, 2004).

Previously we mentioned the benefits of both Agile-based and reuse-based software development. It is not clear however, how these two engineering approaches can be carried out in tandem and very little literature exists on this specific issue. It would be desirable to employ Agile principles to produce simple clear software which is easily adaptable to changing requirements while also employing reuse techniques to improve the software quality and reduce development effort, time and cost. We introduce the term *Agile Reuse* to describe such an approach. For the following reasons it is our position that Agile Reuse is possible and indeed makes sense:

- The simple nature of Agile software makes its reuse appealing to developers. Software is produced in small increments and these small units of software may actually be more reusable than software developed under traditional rigorous methodologies.
- Agile development advocates quick frequent releases of working code. Reuse will help to achieve this.
- Agile developers refactor their code on a regular basis and these very skills are ideal for integrating and tailoring reusable components to match specific needs.

However, implementing an Agile Reuse program is likely to be problematic due to the contradictory principles of both approaches. Table 1 displays a sample of the difficulties that may be encountered.

As a viable solution to the Agile Reuse challenges discussed in Table 1, we propose the use of AI techniques. Based on experimentation, as detailed in Section 6, we have demonstrated that Java classes can be clustered based on their usage of particular software components. We employ collaborative and content-based filtering techniques, as described in Section 5.3, to recommend a set of candidate reusable components to a developer for a particular active class under development. Recommendations for the active class are based on examining similar classes to this class from within the active class' cluster. The use of a recommender agent, RASCAL, which continuously looks for reusable components that the developer may be interested in, is advantageous for a number of reasons as detailed below. These points address the issues raised in Table 1

1. As the developer writes code our agent is always searching for reusable components. Newly developed Agile components do not need support documentation or commenting for our agent to locate or recommend them. These components just need to have been employed at some stage. Based on the context of such employment, our agent will be able to determine when this component is suitable for recommendation. No additional developer effort is required, which is extremely important in an Agile environment.
2. Developers need not initiate the process of component search and retrieval. Instead our agent technology automatically recommends or delivers a suitable component to reuse. We believe component delivery will enhance, promote, and increase the feasibility of reuse to Agile developers as they can quickly and easily employ reusable components and thus produce working code quickly.

Table 1. Agile reuse challenges

No.	Technique	Principle/Belief	Challenge
1	Agile	Working software is the primary measure of success. Less emphasis is placed on design or support documentation and quite often the source code is the only available documentation	Reuse relies on support documentation. Locating an undocumented component is problematic; attempting to reuse this component can be daunting and unappealing to a developer
2	Agile	Customer satisfaction is the main priority. This is achieved through early and continuous delivery of working code.	The developer is focused on producing small working units of software as early as possible. If effective reuse support tools do not exist then a developer will perceive the time taken to locate a reusable component as too costly and a burden to achieving their overall goal
3	Agile	Simplicity is essential	Software developed with simplicity in mind will often tend to be very domain specific and perhaps not as reusable as software developed for a more general or abstract task
4	Reuse	Reuse should be planned	This involves planning the design of highly reusable components. This is time consuming and goes against the grain of Agile development, in particular the principle of simplicity

3. Intuitively, the authors believe that small simplistic units of software developed under Agile methodologies may in fact be more reusable or at least more appealing to reuse than software developed under more traditional heavy processes. The simplicity of Agile components will foster their reuse. Agent technologies will help to support and encourage such reuse which otherwise may

not have occurred. Despite their simplicity, some components may still be initially challenging to understand and integrate with existing work. Our recommender agent produces a recommendation for a component to the active class by examining similar classes which employ this component. Code snippets taken from the similar classes would be a very effective replacement for support documentation which often does not exist with Agile components. Research has shown that developers prefer code examples of components as opposed to descriptive texts (Yunwen and Fischer, 2002).

4. The use of agent technology will not directly produce more highly reusable components. However reuse is intuitive and developer willingness to reuse is evident by the fact that reuse has always existed in software development to some degree, even with insufficient tool support. We believe our RASCAL agent will help encourage high reuse of existing components and as a spin off will help encourage developers to develop new components with reuse in mind.

The potential benefits of an Agile Reuse strategy should be clear. We have listed certain challenges which hamper such reuse and how the use of automatic retrieval technologies can help to overcome such challenges. In the following sections, we will describe in greater detail the overall working of our recommender system and explain the implementation algorithms used.

4. System Overview

As illustrated in Figure 2 our system consists of four components: the active user, the code repository, the usage history collector and the recommender agent. A description of each component is given below. Our current implementation language is Java, but RASCAL is easily adaptable to any OO programming domain.

4.1. *Active user*

The active user can be defined as the developer of the current active class or the current active class itself; the distinction will be clear from the context of the discussion. A class is active if it is currently under development, we assume there to be only one active class at a time. When monitoring user preferences we only consider the usage history

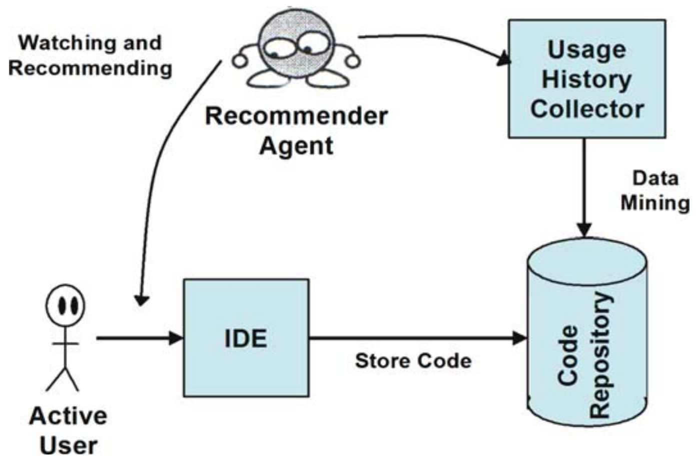


Figure 2. System overview.

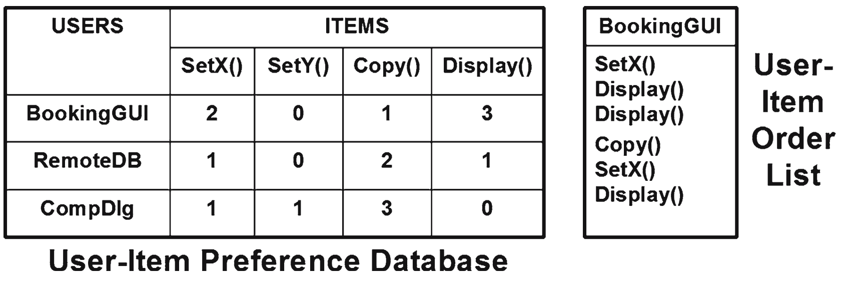
of the current active class and not any other classes this developer may have previously written. A user rating or preference for a software component is based on frequency of use of that particular component (item).

4.2. *Code repository*

Many organisations will maintain a repository of code from previous projects, external libraries, open-source projects, etc. This repository will be continually updated as new classes/systems are developed. All classes developed by the active user are added to the code repository. Code repositories contain a wealth of valuable information. The repository is effectively a user preference database, where a user is a Java class and the components employed by a class are items. In our work, we used the open-source Sourceforge (OSTG, 2004) repository.

4.3. *Usage history collector*

The usage history collector automatically mines the code repository to extract usage histories for all users, which are actually Java classes. This will need to be done once initially for each class and subsequently when a class is added to the repository. We extract this information using the *Bytecode Engineering Library* (Apache, 2003). Component usage histories for all the users are then transformed into a user-item preference database, as shown in Figure 3. We can use



USERS	ITEMS			
	SetX()	SetY()	Copy()	Display()
BookingGUI	2	0	1	3
RemoteDB	1	0	2	1
CompDlg	1	1	3	0

BookingGUI
SetX()
Display()
Display()
Copy()
SetX()
Display()

User-Item Preference Database
User-Item Order List

Figure 3. Sample user-item database.

this database for various tasks such as establishing similarities between users and inferring relationships between components such as order of use. Currently our database contains a user-item preference matrix for all users. Also for each individual user we store a list of components based on their actual usage order. The latter information will be used in Equation (5) in Section 5.2.

4.4. *Recommender agent*

The agent actively monitors the active user and is constantly updating the user preferences. The agent attempts to establish a set of users who are similar to the active user by searching the user-item matrix produced by the usage history collector. The agent then recommends a set of ordered components to the current user. The actual recommendation technique and the technique for measuring the similarity of two users is described in the following section.

5. Recommending Components

A distinction between our recommender system and most mainstream recommenders is that we are trying to predict, in order, the next likely items a developer will employ. Many typical recommender systems only predict a vote for items which the user has not yet tried. Our aim is to predict the next software components to use and it is quite likely that the developer will have used or voted on many of these items in the past. In our approach a recommendation set is produced primarily using collaborative filtering. We also make use of a content-based filtering approach to order this recommendation set. The component which we believe to be most useful to the current developer at this time will appear first in the recommendation set. Below we detail collaborative filtering and the algorithms used to produce our initial recommendation set. We follow this with an overview of the principles of content-based filtering and the algorithm used. We conclude by explaining how we have integrated collaborative and content-based filtering.

5.1. *Collaborative Filtering*

The goal of a collaborative filtering (CF) algorithm is to suggest new items or predict the utility of a certain item for a particular user based on the user's previous preference and the opinions of other like-minded users (Sarwar et al., 2001). The CF 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. In the context of this paper, a user can be considered a Java class and an item refers to a software component and more specifically a Java method.

The CF systems are often distinguished by whether they operate over explicit versus implicit votes. *GroupLens* (Resnick et al., 1994), the first recommender system based on CF to automate predictions, is an example of an explicit voting system. The user consciously votes for items on a scale of one (bad) to five (good). Our developed system uses implicit voting; we automatically deduce the user vote for an item by monitoring the user's frequency of usage of that item.

Breese et al. (1998) classified CF algorithms into two main classes, *Memory-Based* algorithms and *Model-Based* algorithms. Memory-based algorithms operate over the entire user database to make predictions. In contrast, model-based algorithms use the user database to learn a model which is then used for recommendations. Memory-based methods are simpler, seem to work reasonably well in practice and new data can be added easily. For these reasons we decided to use a memory-based algorithm. The following section details the recommendation algorithm developed by Breese et al. (1998); we will extended this algorithm in Section 5.3.

5.1.1. *Recommendation algorithm*

Recommendations are produced from the database of user-item preferences; these preferences are actually a usage count of the item for each user. The user preference database consists of a set of votes v_{ij} corresponding to the vote by user i for item j. The mean vote for user i is calculated as follows:

$$\overline{v}_i = \frac{1}{|I_i|} \sum_{j \in I_i} v_{i,j} \tag{1}$$

where I_i is the set of items the user i has voted on. The predicted vote for the active user a on item j, p_{aj}, is a weighted sum of the votes of the other similar users:

$$p_{aj} = \overline{v}_a + N \sum_{i \in kNN} w_{a,i} \left(v_{i,j} - \overline{v}_i \right) \tag{2}$$

where weight $w_{a,i}$ represents the correlation or similarity between the current user a and each user i. kNN is the set of k nearest–neighbours to the current user. A neighbour is a user who has a high-similarity value, $w_{a,i}$, with the current user. The set of neighbours is sorted in descending order of weight. For experiments, we used a value of $k = 8$. N is the normalising factor such that the absolute values of the weights sum to unity. From Equation (2) we can now predict a user's vote for any item in the user-item preference database. The standard techniques used to calculate $w_{a,i}$ are briefly discussed below (for further explanation see Breese et al., 1998).

Correlation. This is a simplistic technique for calculating similarities between users. The correlation between users a and i is

$$w_{a,i} = \frac{\sum_j (v_{a,j} - \overline{v}_a)(v_{i,j} - \overline{v}_i)}{\sqrt{\sum_j (v_{a,j} - \overline{v}_a)^2 \sum_j (v_{i,j} - \overline{v}_i)^2}} \tag{3}$$

where the summations over j are over the components that **both** users a and i utilised.

Vector similarity. Each user is treated as a vector; the vector holds a count for all components that a user can invoke. The count will hold a value of zero if the user has never used the particular method. The similarity between two users can be computed by determining the cosine of the angle formed by their vectors. This weight is now

$$w_{a,i} = \sum_j \frac{v_{a,j}}{\sqrt{\sum_{k \in I_a} v_{a,k}^2}} \frac{v_{i,j}}{\sqrt{\sum_{k \in I_i} v_{i,k}^2}} \tag{4}$$

where the squared terms in the denominator is used to ensure users who use many components do not appear more similar to other users.

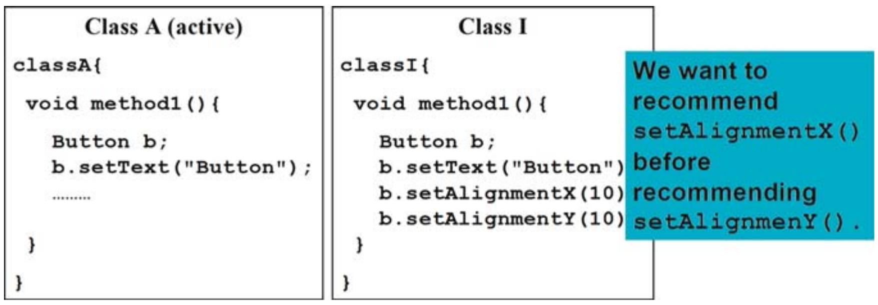
Inverse user frequency. The purpose of this technique is to reduce weights for commonly used components such as standard Java methods `toString()` or `parseInt()`. The motivation for this technique is the belief that items universally liked are less beneficial in capturing similarity than less liked items. The frequency of use for item i, f_i is defined as $\log \frac{n}{n_i}$ where n_i is the number of users who voted for item i and n is the total number of users in the user preference database. We can apply inverse user frequency to the vector similarity technique in Equation (4) by simply transforming the user vote to include the item frequency; for example, $v_{a,j}$ is replaced with $v_{a,j} f_j$.

5.2. *Content-based filtering*

Like CF, the goal of content-based filtering (CBF) (Oard and Marchionini, 1996) is to suggest or to predict the utility of certain items for a particular user. The 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 movie recommender system we would analyse the movies that the current user has rated perhaps analysing movie genre to recommend movies of a similar genre; (Wasfi, 1999) and (Chesnais et al., 1995) present examples of content-based recommender systems. In our work instead of analysing genre, we analyze the order in which components are used.

5.2.1. *Recommendation algorithm*

Content-based filters usually examine item properties such as keywords or genre in order to calculate the similarity between this item and the user's preferences and hence to determine the predicted vote. In our recommender system, we examine the order in which the current user has employed particular items and use this as a basis for calculating a predicted binary vote. Our CBF technique is not intended to be used in isolation but rather it is intended to complement the collaborative approach discussed earlier. The algorithm discussed below is not pure CBF as we examine the neighbour users established using CF. However in principle we could learn content information (such as component ordering) by examining the preferences of random users or all users. For example, for each component in our repository we could store a list of components that frequently precede and succeed uses of that component. Implementing such an approach may be computationally



Class A (active)

```
classA{
 void method1(){
   Button b;
   b.setText("Button");
   .........
 }
}
```

Class I

```
classI{
 void method1(){
   Button b;
   b.setText("Button")
   b.setAlignmentX(10)
   b.setAlignmentY(10)
 }
}
```

We want to recommend `setAlignmentX()` before recommending `setAlignmenY().`

Figure 4. Component ordering.

expensive, which is an important consideration when developing a real-time recommender. We intend to investigate this further.

As stated in Section 4.3, for each user we have stored a list of the actual usage order of items. For example we can determine that user a used item x followed by item y. When making a content-based prediction for item j to the current user a, we examine the last item the current user a used by looking at the user's associated item-usage list, $Prev_a$. We also examine the item previous to item j that user i employed, we will refer to this is $Prev_{i,j}$. If item $Prev_a$ is identical to item $Prev_{i,j}$ then intuitively it is likely that the current user will want to use item j next; indeed it may even be necessary. From Figure 4, when making a prediction for the current user a on the item `setAlignmentX()` which is used by user i, we check to see if the last item employed by the current user (`setText()`) is the same as the previous item user i used before item `setAlignmentX()`. These two items are the same so we would expect to recommend `setAlignmentX()` next. Note using pure CF `setAlignmentX()` and `setAlignmentY()` would be recommended equally with the same predicted vote. Our CBF technique ensures `setAlignmentX()` is recommended first. This technique can be extended to look back for the longest item-usage pattern similarities but for simplicity in our current system we only look back one item from item j. We now define $b_{a,i,j}$ which is the predicted vote using CBF:

$$b_{a,i,j} = \begin{cases} 1 & \text{if } Prev_a = Prev_{i,j} \\ 0 & \text{if } Prev_a \neq Prev_{i,j} \end{cases} \tag{5}$$

where a is the current user, i is a neighbour user from a user preference database and j is the item we are currently making a prediction for.

5.3. *Integrating collaborative and content-based filtering*

The combining of collaborative and CBF is not a novel idea (Alspector et al., 1998; Claypool et al., 1999) but it is particularly suited to our problem domain. Having established the set of k-nearest–neighbours for an active user using the similarity algorithms discussed earlier we now calculate the predicted vote for all items that their nearest–neighbours invoked using Equation (2). However, we extend this equation to take into account the predicted vote determined using CBF. The content-based predicted vote can be considered a bonus value given for item-order similarities. Our final equation for calculating the predicted vote for the active user a on item j, p_{aj}, is as follows

$$p_{aj} = \overline{v}_a + N \sum_{i \in kNN} \left(\left(w_{a,i} \right) \left(v_{i,j} - \overline{v}_i \right) + \left(b_{a,i,j} \right) \left(wcbf \right) \right) \qquad (6)$$

where $wcbf$ is the predefined weight given to the content-based prediction value. In our experiments, we use $wcbf = 0.50$. Referring back to Figure 4, the above equation ensures both `setAlignmentX()` and `setAlignmentY()` are recommended, however, `setAlignmentX()` will appear before `setAlignmentY()` in the recommendation set.

6. Recommendations-Evaluation and Analysis

6.1. *Outline of experiment*

We have conducted experiments to investigate the similarity algorithm discussed in Section 5.1.1 for determining similarities between users and have ascertained which algorithm leads to the most accurate predictions based on Equation (6).

The component repository used in these experiments contained 761 methods from the standard Java Swing library. Recommendations were made for a total of 228 Java classes (users) taken from 30 GUI applications in SourceForge (OSTG, 2004). This included classes that may have been unique or dissimilar from other classes thus having a negative, though realistic, effect on overall average recommendation accuracy.

In each class several sets of recommendations were made. For example, if a fully developed class used ten Swing methods, then we removed the tenth method from the class and a recommendation set was produced for the developer based on the preceding nine methods. Following this recommendation the ninth method was removed from the class and a new recommendation set was formed for this developer based on the preceding eight methods. This process was continued until just one method remained. Each recommendation set contained a maximum of five methods (methods with the highest predicted vote) as we believe this is a sufficient lookahead for a developer.

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, as shown in Equation (7). This represents the probability that a selected item is relevant.

$$p = n_{rs}/n_s \tag{7}$$

where n_{rs} is the number of relevant items selected and n_s is the number of items selected. An item, or component, is deemed relevant if it is used by the developer. Recall, as shown in Equation (8), is defined as the ratio of relevant items selected to the total number of relevant items. This represents the probability that a relevant item will be selected.

$$r = n_{rs}/n_r \tag{8}$$

where n_{rs} is the number of relevant items selected and n_r is the number of relevant items.

6.3. *Analysis*

6.3.1. *Experiment 1*

Figure 5(a) compares the prediction accuracy for each of the three similarity measure techniques plus a baseline result. The baseline recommendations were produced by recommending the top five used methods (ignoring method signature) at each recommendation stage. The graphed results indicate the likelihood that the next method the user will actually employ being in the recommendation set. The next method can be determined by looking back at the original Java class.

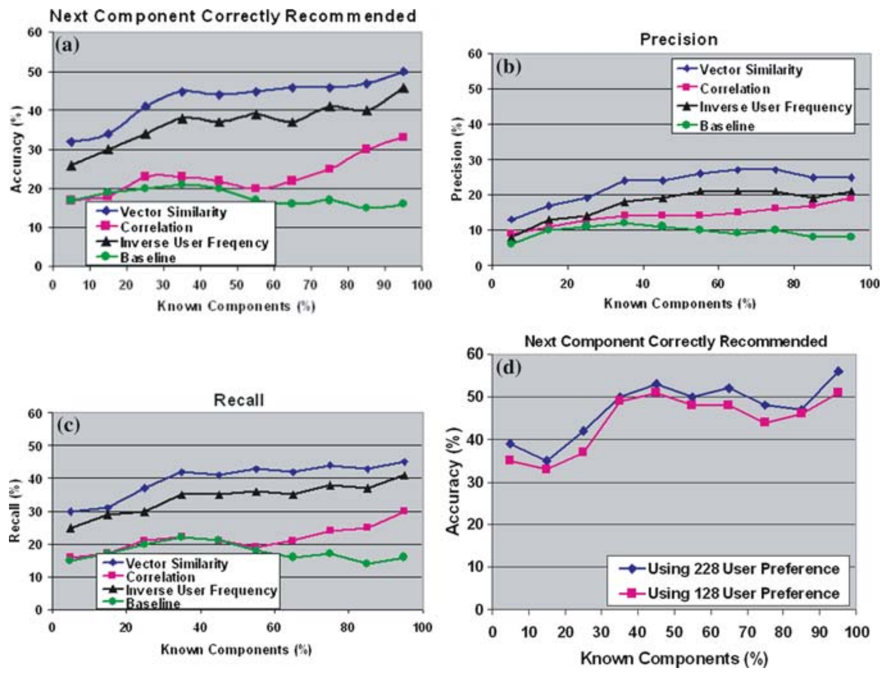


Figure 5. (a) Next component recommended (b) Precision (c) Recall (d) Size of user-item preference database V recommendation accuracy.

For the *Vector Similarity* technique, on average we will correctly predict the next method the developer will use 32% of the time when the developer has only utilised 10% or less of the total methods he/she will actually employ. This encouraging result improves as the developer has utilised more methods. For all recommendations based on *Vector Similarity* there is a 43% likelihood that we will correctly recommend the next method the developer would use. *Vector Similarity* outperforms all of the other techniques at all stages of recommendations. Surprisingly, *Inverse User Frequency* performs poorly compared with *Vector Similarity* which suggests universally liked items are actual useful for establishing user similarities in our examples. Figure 5(b) and (c) display the precision and recall values for each technique. Based on the *Vector Similarity* technique; *recall* was on average 39.8% and average *precision* was 22.7%. When compared to our baseline, these promising results illustrate RASCAL's potential as an effective component recommender.

6.3.2. *Experiment 2*
We conducted an experiment to investigate the importance of the number of user preferences collected. We produced recommendations for 50 users based on the 228 user-item preferences collected. Following this we then produced recommendations for the same 50 users based on a smaller user-item preference database of 128 users after removing 100 random user-item preferences. Both recommendations are based on the *Vector Similarity* technique. The results are displayed in Figure 5(d). As shown, recommendations are marginally better when the larger user preference database is used. This suggests that there is a relationship between the number of user preferences collected and recommendation accuracy. However this relationship is not significant and it does not warrant having an extremely large user preference database which would have a negative effect on recommendation response times. Further experiments are needed to determine the optimum number of user preferences.

7. Conclusions

In this paper, we have proven that Java classes, like humans, can be clustered based on their usage of, or preference for, particular items. We introduced the concept of Agile Reuse and identified specific issues which hamper such reuse. In addressing these issues, we proposed a collaborative content-based recommendation technique to recommend an ordered set of reusable software components to a developer. We also compared three techniques used in CF to establish the similarity between two Java classes and found vector similarity to be most effective. Our recommendation scheme addresses various shortcomings of previous solutions to the component retrieval problem; RASCAL considers the user context and problem domain but uniquely does not place any additional requirements on the developer. The RASCAL is the first recommender system designed specifically to support Agile Reuse.

From experimental results, we have shown RASCAL to be an effective reuse support tool. Opportunities exist to expand RASCAL's scope though. At present, only Java Swing components are recommended but we believe RASCAL is capable of being developed into a general recommender; recommending varied types of software components. However, the number of reusable components in a repository is potentially huge and developing such a system would be complex.

Concept Analysis (CA) (Snelting and Tip, 2000) is one possible solution to developing such a large scale recommender system. The CA could be used to categorise users; for example there may be a category for database component users and another for GUI component users. Once categorised, a user-preference database for each type of user could be established. During the actual recommendation process CA could be used to categorise the class that the developer is currently writing and hence determine the most suitable user-preference database to use for producing recommendations. At this point, a set of candidate software components can be recommended to the current developer using the recommendation techniques discussed in this paper.

RASCAL offers unsolicited advice and we must be sensitive to this in our delivery of recommendations. All recommendations to date have been validated in an automated fashion. If recommendations are to be presented to a developer for validation in realtime, as in a real software tool, then user trials will be needed. Factors such as acceptable recommendation times, component delivery and component integration should all be considered. It may also be necessary to explain how recommendations were derived and give a confidence value for each recommended component. The development of an non-intrusive Eclipse plug-in could be a feasible approach to implementing a real-time recommender. Recommended components could complement the existing context-sensitive list of methods recommended by the Eclipse IDE.

The AI techniques have proven effective in producing recommendations for a Java class to a developer. Our technique produces a recommendation based on Java classes similar to the current class under development. This technique has worked well. It may be beneficial to considering different granularities of users apart from an entire Java class. Recommendations could then be based on methods similar to the current method under development or indeed a hybrid approach could be implemented which would consider both method and class level similarities.

Recommender systems are a powerful technology that can cheaply extract additional knowledge for a software company from its code repositories and then exploit this knowledge in future developments. We have demonstrated that RASCAL offers real promise for allowing developers to discover reusable components with minimal effort. When relatively little information is known about the user we can make reasonably good predictions and our future work will likely strengthen these recommendations. We believe RASCAL will aid developers whilst improving their productivity, enhancing the quality of their code and promoting software reuse.

Acknowledgements

The authors wish to express a sincere thanks to the 15th Artificial Intelligence and Cognitive Science Conference (AICS 2004) reviewers for their constructive and detailed comments. Funding for this research was provided by the Irish Research Council for Science, Engineering and Technology (IRCSET) under grant RS/2003/127.

References

Alspector, J., Kolcz, A. & Karunanithi, N. (1998). Comparing Feature-based and Clique-based User Models for Movie selection. In *Proceedings of the third ACM conference on Digital libraries*, ACM Press, 11–18.

Apache. (2003). Apache Software Foundation – Bytecode Engineering Library (2002–2003). http://jakarta.apache.org/bcel/index.html.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co. Inc.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. (2000). Agile Mainfesto. http://www.agilealliance.org/principles.html.

Bezos, J. (2004). Amazon.com PLC (1995–2004). Seattle, WA 98108-1226, USA http://www.amazon.co.uk.

Breese, J. S., Heckerman, D. & Kadie, C. (1998). Empirical Analysis of Predictive Algorithms for Collaborative Filtering. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, 43–52.

Carroll, J. M. & Rosson, M. B. (1987). Paradox of the Active User. In Carroll, J. M. (ed.) *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. Bradford Books/MIT Press, Chapt. 5, 80–111.

Chesnais, P. R., Mucklo, M. J. & Sheena, J. A. (1995). The Fishwrap Personalized News System. In *IEEE Second International Workshop on Community Networking Integrating Multimedia Services to the Home*.

Claypool, M., Gokhale, A., Miranda, T., Murnikov, P., Netes, D. & Sartin, M. (1999). Combining Content-Based and Collaborative Filters in an Online Newspaper.

Cusumano, M. & Yoffie, D. (1999). Software Development on Internet Time. *IEEE Computer* 32(10), 60–69.

Daudjee, K. S. & Toptsis, A. A. (1994). A Technique for Automatically Organizing Software Libraries for Software Reuse. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, 12.

Drummond, C. G., Ionescu, D. & Holte, R. C. (2000). A Learning Agent that Assists the Browsing of Software Libraries. *IEEE Transactions Software Engineering* **26**(12): 1179–1196.

Frakes, W. & Terry, C. (1996). Software Reuse: Metrics and Models. *ACM Comput. Surv.* **28**(2): 415–435.

Girardi, M. & Ibrahim, B. (1995). Using English to Retrieve Software. *Journals of Systems and Software* **30**(3): 249–270.

Griss, M. L. (1991). Software Reuse at Hewlett-Packard. In Frakes W. (ed.) *In Proceedings of the 1st International Workshop on Software Reusability*. Dortmund, Germany.

Hooper, J. & Chester, R. (1991). *Software Reuse and Methods*. New York: Plenum Press.

Inoue, K., Yokomori, R., Fujiwara, H., Yamamoto, T., Matsushita, M. & Kusumoto, S. (2003). Component Rank: Relative Significance Rank for Software Component Search. In *Proceedings of the 25th international conference on Software engineering*, IEEE Computer Society, 14–24.

Mili, A., Mili, R. & Mittermeir, R. T. (1998). A Survey of Software Reuse Libraries. *Annals of Software Engineering* **5**: 349–414.

Movielens, (2004). GroupLens Research at the University of Minnesota. http://www.movielens.org.

Oard, D. & Marchionini, G. (1996). A Conceptual Framework for Text Filtering Process. Technical report, University of Maryland, College Park.

Ohsugi, N., Monden, A. & Matsumoto, K. (2002). A Recommendation System for Software Function Discovery. In *Proceedings of the 9th Asia-Pacific Software Enginieering Conference (APSEC2002)*.

OSTG (2004). SourceForge.net is owned by the Open Source Technology Group Inc (OSTG), a subsidiary of VA Software Corporation. http://sourceforge.net.

Page, L., Brin, S., Motwani, R. & Winograd, T. (1998). The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project.

Prieto-Diaz, R. & Freeman, P. (1987). Classifying Software for Reuse. *IEEE Software* **4**(1): 6–16.

Raymond, E. (2004). The Cathedral and the Bazar. *http://www.tuxedo.org/esr/writings/cathedral-bazar*.

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

Sarwar, B. M., Karypis, G., Konstan, J. A. & Reidl, J. (2001). Item-based Collaborative Filtering Recommendation Algorithms. In *World Wide Web*. 285–295.

Schmidt, D. (1999). Why Software Reuse has Failed and How to Make it Work for You. *The C++ Report Magazine*.

Sindre, G., Karlsson, E. & Staalhane, T. (1993). A Method for Software Reuse Through Large Component Libraries. In *Proceeding of the International Conference on Computing and Information*. 464–468.

Snelting, G. & Tip, F. (2000). Understanding Class Hierarchies Using Concept Analysis. *ACM Transactions Programing Language Systems* **22**(3): 540–582.

Sugumaran, V. & Storey, V. C. (2003). A Semantic-based Approach to Component Retrieval. *SIGMIS Database* **34**(3): 8–24.

Theunissen, W. H. M., Kourie, D. G. & Watson, B. W. (2003). Standards and Agile Software Development. In *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, South African Institute for Computer Scientists and Information Technologists, 178–188.

Tirso, J. R. (1991). IBM Reuse Program. In *Proceedings of the 4th Annual Workshop on Software Reuse*, University of Maine, 1–5.

Turk, D., France, R. & Rumpepe, B. (2002). Limitations of Agile Software Process. In *Proceedings of Third International Conference on eXtreme Programming and Agile Processes in Software Engineering*, 43–46.

Wasfi, A. M. A. (1999). Collecting User Access Patterns for Building user Profiles and Collaborative Filtering. In *Proceedings of the 4th international conference on Intelligent user interfaces*, ACM Press, 57–64.

Yao, H. & Etzkorn, L. (2004). Towards a Semantic-based Approach for Software Reusable Component Classification and Retrieval. In *Proceedings of the 42nd annual Southeast regional conference*, ACM Press, pp. 110–115.

Yongbeom, K. & Stohr, E. (1998). Software Reuse: Survey and Research Directions. *Management Information Systems* **14**(4): 113–147.

Yunwen, Y. & 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, 159–166.