An Eclipse Plugin to Support Agile Reuse*

Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick

Department of Computer Science, University College Dublin,
Belfield, Dublin 4, Ireland
{frank.mccarey,mel.ocinneide,nick}@ucd.ie

Abstract. Reuse in an Agile context is largely an unexplored research topic. On the surface, these two software engineering techniques would appear to be incompatible due to contradictory principles. For example, Agile components are usually accompanied with little or no support materials, which is likely to hamper their reuse. However we propose that Agile Reuse is possible and indeed advantageous.
We have developed an Eclipse plug-in, named RASCAL, to support Agile Reuse. RASCAL is a recommender agent that infers the need for a reusable component and proactively recommends that component to the developer using a technique consistent with Agile principles. We present the benefits and the challenges encountered when implementing an Agile Reuse tool, paying particular to attention to the XP methodology, and detail our recommendation technique. Our overall results suggest RASCAL is a promising approach for enabling reuse in an Agile environment.

1 Introduction

The demand for organisations to produce new or enhanced software implementations quickly in response to an ever-changing environment has fuelled the use of Agile processes, with Extreme Programming (XP) [1] perhaps the best known and most widely-used Agile methodology. Reuse of software components is another popular software engineering practice. Software reuse has proven to be an effective means of reducing development time and costs whilst benefiting the overall quality of the software [2, 3]. It is not clear however how Reuse and Agile 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. In practice several inherent difficulties arise when considering the compatibility of Agile and reuse techniques due to differences, often contradictory, in their fundamental principles. For example Agile software tends to be simple and domain specific accompanied with minimal support documentation. Reuse relies on support documentation and favors more generalised components.

* Funding for this research was provided by IRCSET under grant RS/2003/127

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 162–170, 2005.

In addition to the above challenges, several other factors hamper reuse independent of the development process used. 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. Often the effort and time taken to locate and integrate reusable components will be perceived to be costly and to outweigh any potential reuse benefits. Indeed, the reality of strict schedules and tight deadlines may mean a developer has simply not the time to search for components; Frakes *et al.* [4] document other barriers to reuse.

In response to these challenges, various intelligent component retrieval techniques have been developed to assist a developer discover or locate components in an efficient manner [5]. These techniques share a common shortcoming though; the developer must initiate the retrieval process. In our work, we shift the attention from component retrieval to component recommendation. We have developed a recommender tool, named RASCAL, for software components. RASCAL has been developed for two purposes. Firstly we wish to recommend software components that the developer is interested in. Secondly, and more importantly, we wish to recommend useful components which the developer may not be familiar with or aware of. We believe recommendations will assist and encourage developers in making full use of large component repositories in an efficient manner and in turn will help to promote software reuse. Our work is geared towards supporting Agile Reuse, paying particular attention to XP. The goal of RASCAL then is to recommend useful components to a developer in a way which is consistent with the principles of XP 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.

In this paper we introduce Agile Reuse, present our support tool RASCAL and explain the AI recommendation technique employed. An overview of RASCAL's implementation is given in the following section. In section 3 we detail Agile Reuse; we discuss the benefits of such reuse in an XP context and identify the difficulties of providing an XP tool to support this concept. Two recommendation techniques are discussed in section 4; we then present our hybrid approach followed by a short analysis of the experimental results. In section 5 we review related work in the area of component search, retrieval and recommendation. Finally we discuss how RASCAL can be extended and draw general conclusions in section 6.

2 System Overview

RASCAL is implemented as a plug-in for the Eclipse IDE, as illustrated in figure 1. As a developer is writing code, RASCAL monitors the methods currently invoked and uses this information to recommend a candidate set of methods to this developer. Recommendations are then presented to the developer in the recommendations view at the bottom right hand corner of the IDE window. Currently, RASCAL recommends methods from the Swing and AWT toolkits.

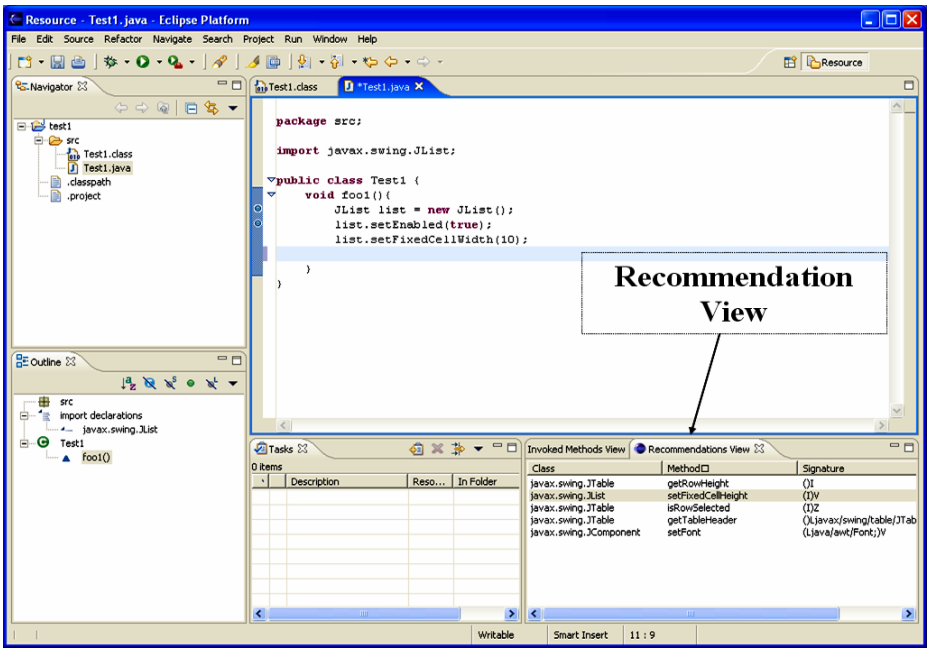
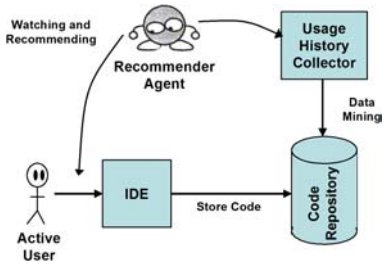


Fig. 1. Eclipse Prototype

Figure 2(a) display a general overview of our system which consists of four components: the *active user*, the *code repository*, the *usage history collector* and the *recommender agent*. 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. When monitoring user preferences we only consider the usage history of the current active class and not any other classes this developer may have previously written. The code repository maintains code from all previous projects and all newly created classes will be added to this repository. In our work, we built a code repository using open-source software available from *Sourceforge* [6].

The usage history collector automatically mines the code repository to extract component usage histories for all the stored Java classes. This will need to be done once initially for each class and subsequently when a class is added to the repository. Component usage histories for all the users are then transformed into a user-item preference database, as shown in figure 2(b), which can be used to establish similarities between users. Also, for each individual user we store a list of components based on their actual usage order. The latter information is used for Content-Based filtering as discussed in section 4.1. Finally the recommender agent actively monitors the Java class that the developer is coding, noting in particular the components used in this class. The agent attempts to establish a set of neighbouring users who are similar to the active user by searching the user-item preference database. A set of ordered Java methods is then recommended to the active user based on the neighbouring users.



(a)

USERS	ITEMS			
	SetX()	SetY()	Copy()	Display()
Hotlist	2	0	1	3
RemoteD	1	0	2	1
CompDlg	1	1	3	0

User-Item Preference Matrix

Hotlist
SetX()
Display()
Display()
Copy()
SetX()
Display()

User-Item Order List

(b)

Fig. 2. (a) System Overview (b) Sample user-item database

3 Agile Reuse

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. Ten reusable aspects of any given software project are presented by Frakes *et al.* [4] in their discussion of reuse metrics and models, including 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 [3] and time to market [2].

Despite these desirable advantages several factors hamper reuse as discussed in the introductory section. Factors vary from technical difficulties such as support environments to more pragmatic issues such as managerial and developer attitudes. As reuse becomes more prominent and accepted in industry, systems and tools that aid and support reuse become key aspects in achieving successful reuse of software artifacts [7]. 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 [5].

We have mentioned the benefits of reuse-based software development, however, it is unclear how this software engineering approach can be carried in tandem with Agile development. There is an absence of literature and tools to support this concept. 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 describe such an approach as *Agile Reuse*. Our work focuses on the technical issues involved in implementing this approach; we pay particular attention to Agile Reuse in an XP environment though the issues raised are relevant to all Agile processes. For the following reasons it is the authors position that Agile Reuse using XP is possible and indeed makes sense:

- The simple nature of XP 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.
- XP development advocates quick frequent releases of working code. Reuse will help to achieve this.
- XP developers refactor their code on a regular basis and these very skills are ideal for integrating and tailoring reusable components to match specific needs.

In practice several inherent difficulties arise when considering the compatibility of XP and reuse techniques due to differences, often contradictory, in their fundamental principles. Table 1 on the following page displays a sample of such difficulties that may be encountered and illustrates why providing tool support for reuse in an XP context is difficult. In addition to this we also explain how our support tool, RASCAL, can be employed to address these issues. In the next section we describe how RASCAL automatically retrieves and recommends components, and present experimental results.

4 Recommendations

4.1 Recommendation Technique

Recommendations are produced using a hybrid of two popular filtering techniques, namely collaborative filtering and content-based filtering. The goal of Collaborative Filtering (CF) algorithms 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 [8]. 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. Like CF, the goal of Content-Based Filtering (CBF) [9] 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 preference are likely to be recommended. For example in a news recommender system we would analyse the keywords from the current user's preference to recommend news stories which contain similar keywords; keywords could be "business" or "sport". In our work, instead of analysing keywords or categories we analyse the order in which components are used. In our hybrid recommendation technique we produce our primary recommendation set using CF. We then make use of CBF to order the initial 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.

4.2 Evaluation

We have conducted experiments to investigate the accuracy of our hybrid algorithm. The component repository used in these experiments contained 1888

Table 1. XP Reuse Challenges and RASCAL

Practice/Belief	Challenge	RASCAL
Working software is the primary measure of success. Less emphasis is placed on comprehensive 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.	As the developer writes code our agent is continually searching for reusable components. Newly developed XP 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.
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.	Developers need not initiate the process of component search and retrieval. Instead RASCAL automatically recommends or delivers a suitable component to reuse. We believe component delivery will enhance, promote and increase the feasibility of software reuse to XP developers as they can quickly and easily employ reusable components and thus produce working code quickly.
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.	We propose that the simplicity of XP components fosters their reuse. RASCAL 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. RASCAL produces a recommendation for a component to a class by examining similar classes which employ this component. Code snippets taken from the similar classes could prove to be an effective addition to the minimal documentation which often accompanies XP components.

methods from the standard Java Swing library and the Abstract Window Toolkit (AWT). Recommendations were made for a total of 508 Java classes (users)

which invoked on average 60 methods. These classes were taken from 60 GUI applications in SourceForge [6].

For each class several sets of recommendations were made. For example, if a fully developed class used 10 Swing methods, then we removed the 10th method from the class and a recommendation set was produced for the developer based on the preceding 9 methods. Following this recommendation, the 9th method was removed from the class and a new recommendation set was formed for this developer based on the preceding 8 methods. This process was continued until just 1 method remained. Each recommendation set contained a maximum of 5 methods as we believe this to be a sufficient lookahead for a developer. We evaluated the results using *Precision* and *Recall* [10]. Precision represents the probability that a recommended method is relevant. Recall represents the probability that a relevant method will be recommended. Based on our repository of original classes, we also evaluate whether the actual next method a particular developer invoked is in our recommendation set. This is an important evaluation as we wish to recommend methods in an realistic and meaningful order.

4.3 Results

Figure 3 displays the results of our recommendation technique. We also present a baseline result based simply on recommending the five most commonly used methods at each recommendation stage. The recommendation precision is displayed in figure 3(a); the average precision of our technique is 20% which compares favorably with our baseline result. Recall is displayed in figure 3(b); the average recall, based on our recommendation algorithm, is 36%. That is, if we were to recommend ten methods, then on average almost four of those recommended methods would be relevant. Finally, in figure 3(c) we display the likelihood that the next method the developer will actually invoked will be in our recommendation set; on average there is 43% likelihood that it will be. Further to this encouraging result, we see that RASCAL can make reasonably accurate predictions at a relatively early stage in the class's development. For example, when a developer has invoked 20% or less of the total methods she will employ then there is 42% likelihood that RASCAL will correctly recommend the next invocation. We only present the results of our hybrid approach here as we have ascertained that this algorithm leads to the most accurate predictions; [11] details the implementation details, benefits and accuracy of the individual CF and CBF algorithms.

5 Related Work

Much research on tool support for software reuse has concentrated on intelligent search and retrieval techniques which are dependent on developer initiation, for example [5]. However, to effectively and realistically support component reuse it is tremendously important that component retrieval be complemented with unsolicited component delivery/recommendation. One technique to address this issue is *CodeBroker* [12]. CodeBroker infers the need for components and proactively

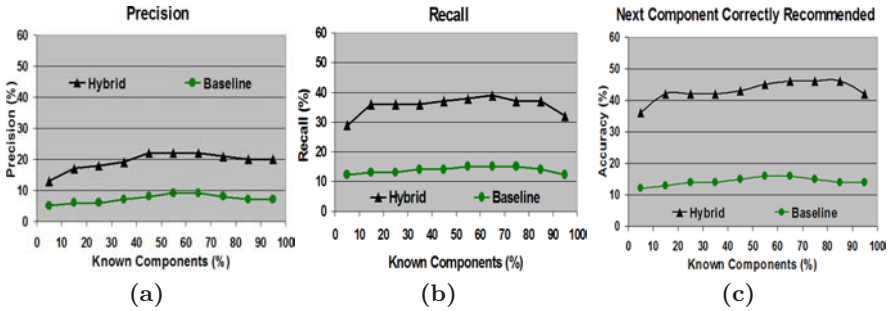


Fig. 3. (a) Precision (b) Recall (c) Next found

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 traditional retrieval approaches, but it does not address the requirements of Agile Reuse. The reusable components in the repository must be sufficiently commented to allow matching, this may exclude many components. 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 XP and other Agile methodologies.

Ohsugi *et al.* [13] 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. For clarity, *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 proposed is an extension of traditional collaborative filtering algorithms used in mainstream recommender systems such as *Amazon*. In our work we apply Ohsugi's principle to a different problem domain, namely reusable software components. Similar to *CodeBroker* [12] 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. Unlike the related works, our technique is specifically designed to assist reuse in an XP environment.

6 Conclusions

In this paper we introduced the concept of Agile Reuse and identified specific issues which hamper such reuse. In addressing these issues, we evaluated collaborative and content-based filtering and found a hybrid approach to be most effective. Our recommendation scheme addresses various shortcomings of previous solutions to the component retrieval problem; user context and problem domain are considered while no additional requirements are placed on the de-

veloper. Opportunities exist to expand RASCAL's scope though. Firstly, we will develop RASCAL into a general recommender capable of recommending various component types. RASCAL will then be extended to allow greater user interaction; for example an accepted recommendation will be automatically added to the user's code. With any unsolicited recommender, delivery is important. Using established industrial links, extensive user trials are planned which we hope will foster a more usable application.

Recommender systems are a powerful technology that can cheaply extract 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 discover reusable components and is well suited to Agile development. When little information is known about the user we can nevertheless make reasonably good predictions and future work will likely strengthen recommendations. We believe RASCAL will aid developers whilst improving their productivity, enhance the quality of their code and promoting software reuse.

References

1. Beck, K.: XP explained: embrace change. Addison-Wesley Publishing Co. (2000)
2. Yongbeom, K., Stohr, E.: Software reuse: Survey and research directions. Management Information Systems **14** (1998) 113–147
3. Hooper, J., Chester, R. In: Software Reuse: Guidelines and Methods. Plenum Press, NY (1991)
4. Frakes, W., Terry, C.: Software reuse: metrics and models. ACM Surv. **28** (1996)
5. Yao, H., Etzkorn, L.: Towards a semantic-based approach for software reusable component classification and retrieval. In: Proceedings of the 42nd annual Southeast regional conference, ACM Press (2004) 110–115
6. OSTG: Open source technology group inc (ostg). `http://sourceforge.net`. (2004)
7. Daudjee, K.S., Toptsis, A.A.: 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 (1994) 12
8. Sarwar, B.M., Karypis, G., Konstan, J.A., Reidl, J.: Item-based collaborative filtering recommendation algorithms. In: World Wide Web. (2001) 285–295
9. Oard, D., Marchionini, G.: A conceptual framework for text filtering process. Technical report, University of Maryland, College Park (1996)
10. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison Wesley, New York (1999)
11. McCarey, F., Cinneide, M., Kushmerick, N.: Knowledge reuse for software reuse. In: Submitted to the 17th International Conference on Software Engineering and Knowledge Engineering. (2005)
12. Yunwen, Y., Fischer, G.: Information delivery in support of learning reusable software components on demand. In: Proceedings of the 7th international conference on Intelligent user interfaces, ACM Press (2002) 159–166
13. Ohsugi, N., Monden, A., Matsumoto, K.: A recommendation system for software function discovery. In: Proceedings of the 9th Asia-Pacific SE Conference. (2002)