Supporting Agile Reuse Through Extreme Harvesting

Oliver Hummel and Colin Atkinson

University of Mannheim, Chair of Software Technology
68159 Mannheim, Germany
{hummel,atkinson}@informatik.uni-mannheim.de
http://swt.informatik.uni-mannheim.de

Abstract. Agile development and software reuse are both recognized as effective ways of improving time to market and quality in software engineering. However, they have traditionally been viewed as mutually exclusive technologies which are difficult if not impossible to use together. In this paper we show that, far from being incompatible, agile development and software reuse can be made to work together and, in fact, complement each other. The key is to tightly integrate reuse into the test-driven development cycles of agile methods and to use test cases - the agile measure of semantic acceptability - to influence the component search process. In this paper we discuss the issues involved in doing this in association with Extreme Programming, the most widely known agile development method, and Extreme Harvesting, a prototype technique for the test-driven harvesting of components from the Web. When combined in the appropriate way we believe they provide a good foundation for the fledgling concept of agile reuse.

1 Introduction

Agile development and software reuse are both strategies for building software systems more cost effectively. Agile methods do this by shunning activities which do not directly create executable code and by minimizing the risk of user dissatisfaction by means of tight validation cycles. Software reuse does this by simply reducing the amount of new code that has to be written to create a new application. Since they work towards the same goal it is natural to assume that they can easily be used together in everyday development projects. However, this is not the case. To date agile development and systematic software reuse have rarely been attempted in the same project. Moreover, there is very little if any mention of software reuse in the agile development literature, and at the time of writing there is only one published reuse concept whose stated aim is to reinforce agile development. This is the so called "agile reuse" approach of McCarey et. al. [12].

The reason for this lack of integration is the perceived incompatibility of agile approaches and software reuse. Whereas the former explicitly eschews the creation of software documentation, the latter is generally perceived as requiring it. And while agile methods usually regard class operations (i.e. methods) as defining the granularity of development increments, reuse methods typically regard classes as the smallest unit of reuse in object-oriented programming. As a third difference, reuse approaches

G. Concas et al. (Eds.): XP 2007, LNCS 4536, pp. 28–37, 2007.

tend to be more successful the "more" explicit architectural knowledge is reused (as in product line engineering), whereas agile development methods employ as little explicit architecture as possible. At first sight, therefore, there appears to be several fundamentally irreconcilable differences between the two approaches.

McCarey et. al suggest a way of promoting reuse in agile development through so-called "software recommendation" technology. Their "agile reuse" tool, RASCAL [10] is an Eclipse plug-in which uses collaborative and content-based filtering techniques [9] to proactively suggest method invocations to developers. It does this by attempting to cluster Java objects according to the methods they use, just as Amazon, for example, clusters its customers according to the books they buy. The tool monitors method invocations in the class currently under development to predict method calls that are likely to be soon needed and suggests them to the developer. To evaluate their system the authors experimentally predicted invocations of the Java Swing Library in common open source systems and claim precision rates of around 30%.

Although the concept of RASCAL fits well into the agile spirit of providing maximum support for "productive" activities, there is nothing in the technology which ties it specifically to agile development. The approach embodied in RASCAL can just as easily be used with any other development methodology that produces code, including traditional heavyweight processes. Moreover, the approach has the same fundamental weakness as other repository-based approaches – the quality of the recommendations is only as good as the quality of the code repository that is used to search for components. Unfortunately, to date there have been few if any successful attempts to set up and maintain viable component repositories [6]. The version of the tool described in [10] is clearly a prototype, but McCarey et el. do not present a strategy for solving this important problem. Moreover, although RASCAL showed impressive performance for the limited domain of Swing invocations, it is not clear whether this technique will work for other domains with many more classes that have much lower usage frequencies.

1.2 The Core Challenge

The core challenge of agile reuse lies in developing a reuse strategy that complements the principles of agile development and offers a way of promoting reuse in tandem with the key artifacts and practices of agile methods. Whilst proactive recommendation technology such as RASCAL is certainly valuable in complementing such a strategy it does not itself solve the issues mentioned above. In this paper we present an approach which we believe does address these challenges and thus represents a viable basis for the concept of agile reuse. The key idea is to use unit test cases, which in most agile methods should be defined before implementations, to influence the component searching process. Such test-driven development is one of the most fundamental principles of Extreme Programming, the most widely used agile method. Tests are used as the basic measure of a unit's semantic acceptability. Once a code unit passes the tests defining its required behaviour it is regarded as "satisfactory" for the job in hand.

Usually the code to satisfy the tests for a unit is implemented by hand. However, there is no specific requirement for this to be so. Since passing the corresponding tests is the measure of acceptability, any code module that passes the tests is functionally acceptable whether created from scratch or retrieved from a component repository. A search technology which can systematically deliver code that passes the tests defined for components will therefore make the implementation of new code unnecessary. In our opinion, the combination of test-driven development and test-driven retrieval, as proposed in a rudimentary form in [11], create a natural synergy between agile development and software reuse and provide a solid foundation for the notion of "agile reuse". Due to its roots in test-driven development we have called our solution "Extreme Harvesting".

The rest of this paper is structured as follows. In the next section we briefly review the key ideas of Extreme Programming and introduce a simple example from a well know book in the field. In the section after that we discuss the difficulties involved in promoting software reuse and introduce the notion of Extreme Harvesting, our test-driven technique for finding components on the Internet and other large scale component repositories. Section 4 then explains how Extreme Harvesting might be used to support software reuse in the context of agile development – so called "agile reuse". Section 5 presents some of the prototype tools that we have developed to explore this approach. Finally, we conclude our contribution in section 6.

2 Extreme Programming Revisited

In this paper we use Extreme Programming as the archetypical example of an agile development method. However, our approach is not limited to Extreme Programming (XP) but is in principle applicable to any other process where tests are written before the implementation as well (like Agile Modeling [2] for example). In this section we briefly highlight those aspects of XP that are of importance for the understanding of our approach. We assume that the reader is familiar with other fundamental principles of Extreme Programming such as the four values of communication, simplicity, feedback and courage and the many recommended practices. For further details we refer to [4], for instance.

The test-driven nature of XP requires in particular that unit tests be written before any code for that unit is developed. These tests are used as the primary measure for completion of the actual code. The maxim is that anything that can't be measured simply doesn't exist [14] and the only practical way to measure the acceptability of code is to test it. To illustrate how test-driven development works in practice let us consider a small example.

```
public class Movie {
    public Movie(String title, int priceCode) {}
    public String getTitle() {}
    public int getPriceCode() {}
    public void setPriceCode(int priceCode) {}
}
```

This class, Movie, offers a constructor with arguments for the title and price code of a movie and methods for accessing (i.e. getting) these values. It also provides a method for setting the price code to a new value. Together with classes Customer and Rental, it represents the initial version of the well-known video store example from Martin Fowler's refactoring book [3]. Beck [14] and others recommend that the methods be developed and tested sequentially. Thus, a typical XP development cycle applied to the class Movies might tackle the methods of the class in the following order -

- Constructor with title and price code
- Retrieve title
- Retrieve price code
- Change price code

The basic idea is to define tests to check that the constructor works correctly in tandem with the retrieval method. This can be done using one combined test or using a separate test for each retrieval method. In this example we choose the latter since it is the more realistic for larger components. Thus, a JUnit [5] test case for the retrieval of the movie's title of the following form is created (usually, the test case is created before the stub in XP, of course):

```
public void testTitleRetrieval() {
       Movie movie = new Movie("Star Wars", 0);
       assertTrue(movie.getTitle().equals("Star Wars"));
}
```

In practice, test cases would probably be more elaborate (for example, they might follow the principle of triangulation [14]) but due to a lack of space we stay with a simple example here. This should be enough to convey the core ideas. In the next step, a stubbed out version of the Movie class (similar to the signature above) with just the constructor and the getTitle method is generated and made to compile. After this, the test case and stub are compiled, and the test is run to verify that a red bar is obtained from JUnit. Once the failure of the test has been checked, the stub is filled with the simplest implementation that could possibly work, and the test is re-run until a green bar is received from JUnit. The to-do list is then updated accordingly:

- ~~Store title and price code~~
- ~~Retrieve title~~
- Retrieve price code
- Change price code

The same process is then applied to the next method(s) on the to-do list until the class as a whole has been completed. After each iteration, some design work may become necessary to refactor the implementation of the class.

3 Software Reuse and Extreme Harvesting

There is more or less general consensus on the two important prerequisites that need to be fulfilled to support the systematic reuse of small and medium-sized components

of the kind typically handled in agile development environments. The first is the availability of a library or so-called repository that stores reuse candidates and the second is a query technique that allows components to be retrieved effectively [7]. At first sight, these may appear to be trivial, but in practice this is not the case. The effort involved in setting up and maintaining a suitable repository is typically so high that some researchers do not expect to see a working solution for a long time to come [6]. There are many examples of projects from the past in which researches and developers have tried to setup and maintain public component repositories of even just a few hundred components but have eventually failed due to the associated maintenance problems. The recent shutdown of the Universal Business Registry for web services is another high profile example of this. Lately a few commercial code search engines have emerged which focus on supporting searches for code on the Internet (e.g. google.com/codesearch, koders.com and merobase.com). These provide various techniques for retrieving components from software libraries but none of them have yet found the right combination of prescriptiveness, ease of use and performance to become widely used. The well-known survey of Mili et al. [1] describes the related problems in more detail.

To integrate a reusable software unit into a system one generally needs the unit's syntactical description (i.e. its interface) and a corresponding semantic description of its functional effects. In Extreme Programming, as with most other object-oriented development approaches, a unit's syntactic interface is represented by its method signatures. Where Extreme Programming differs from most other methods, and what makes it particularly suitable as a basis for systematic reuse, is its provision of a simplified semantic description of the behaviour of a unit's operations before they are actually implemented. Most other methods have no such semantic description of operations, since formal methods (for example based on pre- and post-conditions) have so far proven impractical in mainstream development. These semantic descriptions of operations (i.e. the test cases) in XP are the vital prerequisite for being able to establish whether discovered components are fit for the required purpose. At present, however, only merobase offers full support for intelligent interface-driven searches in which components are retrieved based on the abstractions that they represent rather than on the presence of certain strings in their code. Our Extreme Harvesting approach, however, revolves around the principle of using the test cases defined for a desired component to filter out discovered components which are not fit for the purpose in hand. Figure 1 below provides a schematic summary of the six basic steps involved:

a) define semantics of desired component in terms of JUnit test cases
b) derive interface (i.e. the class stub) of desired component from test cases
c) search candidate components in repository using search term derived from (b)
d) find source units which have signatures matching the one defined in (b)
e) filter out components which are not valid (i.e. compilable) source units
f) establish which components are semantically acceptable by applying the tests defined in (a)

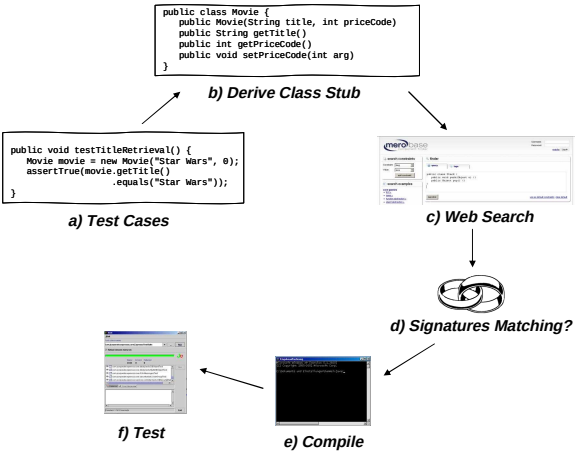

Fig. 1. Schematic process overview

4 Agile Reuse

As explained above, the basic idea behind our notion of agile (or extreme) reuse is to use test cases developed as part of the normal activity of Extreme Programming as the basis to search for suitable existing implementations. However, we believe there are two basic ways in which agile development could benefit from reuse, depending on the goals of the developer and the novelty of the application under development, namely definitive vs. speculative harvesting. In effect these approaches – which we will explain in the next two subsections – occupy opposite ends of a spectrum. At one end we have the situation in which the nature of the reused components is driven by a rigid design as it is typically used in traditional heavyweight approaches. We call it definitive harvesting. At the other end of the spectrum we have the situation in which the nature of the software design is influenced by the reused components what we call speculative harvesting. Practical software reuse activities will usually lie somewhere in between.

4.1 Definitive Harvesting

In projects in which the domain and requirements are well understood, or where the architecture is rigid, the interface and desired functionality of components might be fairly fixed and more importantly fairly common. When this is the case, it makes sense to develop all of the test cases and interface descriptions, for all of the items in the "to do" list, upfront, and then attempt to harvest as many suitable components as possible following the process shown in figure 1 above. If no suitable reuse candidates can be retrieved the required component has to be implemented in the regular way. However, if a reuse candidate can be found (e.g. a search for the interface of our Movie stub from section 2 on merobase delivers 25 candidates), a great deal of effort can be saved. Depending on the type of component and the size of the underlying repository the chances of finding something vary significantly, but since the overall amount of implementation effort (in the event that no suitable reusable component is

found) is the same as in the unmodified case (i.e. in regular extreme programming) there would be no needless work involved.

4.2 Speculative Harvesting

Speculative harvesting, on the other hand, is best when the developer is unsure what the interface and complete functionality of the component should be, and is willing to adapt the rest of his/her system to the interface of any suitable component that may be found. This tends to occur more frequently in (agile) projects in an entirely new domain or with highly innovative characteristics. With this approach the developer creates test cases one after the other, working through the to-do list as recommended by Beck. However, before implementing the functionality to make the class pass the test (as in normal extreme development), a general search can be performed on the methods featured in the test case (e.g. a query for movie and getTitle on a search engine) to see if there are any existing classes from the Internet which might be able to pass it. If there are none, then obviously the developer has no choice but to go ahead and develop the component his/herself. If there are just a few, then the developer can quickly study and evaluate these to see if any of them provides the desired functionality for the whole class. If there are a large number of results, which is often the case when the query is very general, the developer can define the next test case according to the original to-do list and the implementations available. Then another test-driven search can be performed (using both exiting test cases), and the same decision process is applied again. This time, the set of results will be smaller, because by definition the set of components that pass both test cases will be a subset of the set which passed just one. However, if the result set is still very large, the developer can add another test case and continue in this fashion until either a suitable component is found or the set of results has been reduced to a number that can reasonably be analyzed. This process can be summarized as follows -

(1) Develop the next test on the “to do” list
(2) Perform a test driven search for the included functionality using all available tests
(3) See how many results there are

a) None ➔ abandon all searching and implement whole class as normal
b) A few ➔ analyze each class and

(i) if there are any suitable– use one of them
(ii) if not abandon searching and implement as normal

c) A lot ➔ if the “to do” list is not empty, repeat the process from the start; if the “to do” list has been completed, abandon the process and implement whole class as normal

5 Prototypical Tool Support

Currently we are working in two directions to support this vision of Extreme Harvesting with appropriate tools. We have been developing an Eclipse plug-in which is able to harvest components from various code search engines from the Web with just a

single mouse-click on the class stub once the developer has entered the desired method declarations and test cases as shown in figure 1. Proprietary repositories can also be used if they offer an API for programmatic access. For the videostore example, we were able to harvest multiple working implementations of the Movie class (Google: 16, Koders: 2, Merobase: 14) as shown in a screenshot below where our tool is used inside the well-known Eclipse IDE.

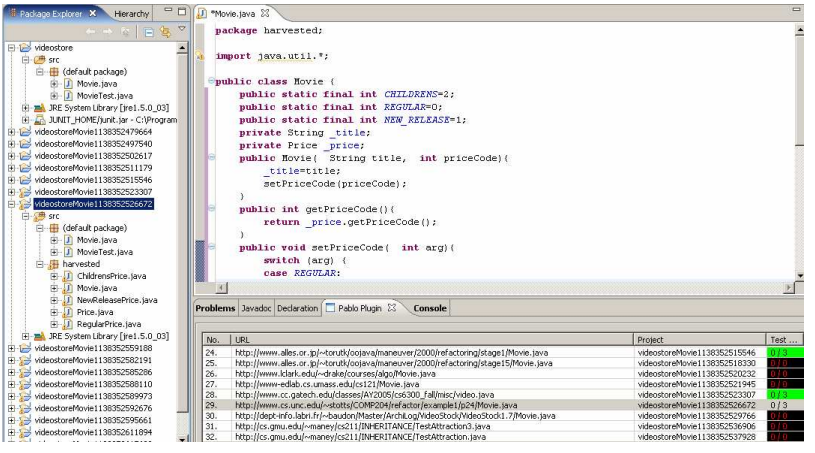

Fig. 2. Except of search results with a positively tested Movie class

The table at the bottom shows the URLs where reuse candidates have been found and the associated test result. A "0/3" with a green background denotes 3 test cases with 0 failures, a red "0/0" on black background is shown for classes that did not match syntactically and hence did not compile. These would not be shown to the user in a full version of the tool, but are currently kept for evaluation purposes. Like the "traditional" JUnit tool, the green "0/3" is the signal that a unit that has passed all tests and hence is appropriate for the required purpose. The editor above the table shows the code of an exemplary harvested class and the package explorer on the left contains all files that have been harvested.

As shown in the figure we obtained a version of Movie which depends on an additional class, Price, to work. Our tool automatically recognized this dependency and retrieved the Price class and its subclasses that were also needed. As mentioned in section 2, Fowler's videostore example in its initial version only involves two additional classes, Customer and Rental, which were also retrieved from the Web using the strategy outlined above. Missing dependencies to other classes can also be traced automatically to a limited extend, but we are currently working to further improve this capability. If the harvesting and the associated upfront "guessing" of the required functionality is successful, after stub and test case definition the only additional step is the invocation of our reuse tool for definitive harvesting. Our tool is also able to automatically create an object adapter [13] if this should become necessary. Further implementation work usually involved in building a component from scratch is thus avoided. Overall, the initial results from this approach are promising. During our

initial experiments, we were able to retrieve multiple ADTs (like Stack etc.), implementations of a class for matrix calculations, a small ensemble of a deck and a card class for a card playing application, as well as various sorting algorithms, mathematical functions and other examples from the reuse literature. These results are described more fully in [8]. Results demonstrating the precision improvement of interface-driven searches and Extreme Harvesting compared with other approaches have recently been submitted elsewhere. Although our prototype currently focuses on Java it should be easily adaptable to other languages or even web services since they all offer a syntactic description of the interface to a piece of functionality and the possibility to test it.

The second prototype we are working on is intended to provide better support for the agile spirit of speculative harvesting. It is shown in figure 3 below. In the case when a developer starts with a very general search like a class movie with just a getTitle method (cf. figure 3) for example he will potentially receive a large number of results. They will all have at least some similarity to the component that the developer needs because they all pass at least one test. From these results it is possible to calculate a kind of "average search result" which collects together the features of the individual results and generates a representative summary. This tool is thus able to recommend canonical operations similar to the operation invocations delivered by RASCAL [10].

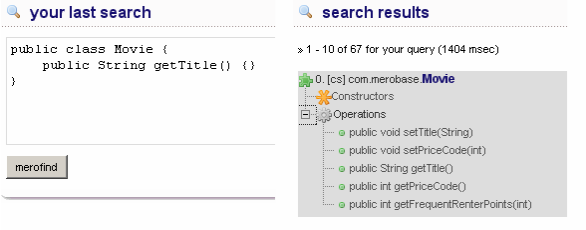

Fig. 3. Screenshot of a query and canonical operation recommendations derived from it

The developer can now choose from these canonical operations to constrain his next search in accordance with the to-do list. In the future this process might even support the creation of the to-do list itself by deriving common functionality for general concepts from a repository. We plan to elaborate on this on another occasion.

6 Conclusion

This paper has addressed the issue of promoting systematic reuse in agile development projects and explained how the use of a test-driven harvesting approach, known as Extreme Harvesting, can overcome the prima facea incompatibilities of the two paradigms. We presented an approach, based on the notion of Extreme Harvesting, which allows reuse and agile development to be combined in a complementary way. In short, the approach allows agile development to be enhanced through systematic reuse, and thus provides a solid basis for the notion of "agile reuse" coined by [12].

Due to this seamless integration in XP it is easy to use Extreme Harvesting in a reactive way (i.e. when the developer must explicitly invoke it) as well as in a proactive recommendation-oriented way in the spirit of RASCAL [10]. We believe that the presented approach therefore complements the work of McCarey et. al. We are currently working on improving our tool to proactively suggest code units that have been selected using automated clustering techniques of the form found in information retrieval research [9] and successfully applied in RASCAL. We hope that such an extension will also make it possible to anticipate possible refactorings in a given (or retrieved) code unit if only enough equivalent units could be retrieved from the repository. Furthermore, we are planning to conduct empirical studies with students to gain more experience about the value of our tool in practical settings.

References

1. Mili, A., Mili, R., Mittermeir, R.: A Survey of Software Reuse Libraries. Annals of Software Engineering, vol. 5 (1998)
2. Ambler, S., Jeffries, R.: Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. John Wiley and Sons, Chichester (2001)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
4. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading (1999)
5. Beck, K., Gamma, E.: JUnit: A Cook's Tour. Java Report (August 1999)
6. Seacord, R.: Software Engineering Component Repositories. In: Proceedings of the International Workshop on Component-based Software Engineering, Los Angeles, USA (1999)
7. Frakes, W.B., Kang, K.: Software Reuse Research: Status and Future. IEEE Transactions on Software Eng., vol. 31(7) (2005)
8. Hummel, O., Atkinson, C.: Using the Web as a Reuse Repository. In: Proceedings of the International Conference on Software Reuse, Torino (2006)
9. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, London, UK (1999)
10. McCarey, F., Ó Cinnéide, M., Kushmerick, N.: An Eclipse Plugin to Support Agile Reuse. In: Proc. of the 6th Int. Conf. on Extreme Progr. and Agile Processes, Sheffield (2005)
11. Podgurski, A., Pierce, L.: Retrieving Reusable Software by Sampling Behavior, ACM Transactions on Software Engineering and Methodology, vol. 2(3) (1993)
12. McCarey, F., Ó Cinnéide, M., Kushmerick, N.: RASCAL: A Recommender Agent for Agile Reuse. In: Artificial Intelligence Review, vol. 24(3-4), Kluwer, Dordrecht (2005)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
14. Beck, K.: Test Driven Development: By Example. Addison-Wesley, London, UK (2002)