A SEMI-AUTOMATED PROCESS FOR OPEN SOURCE CODE REUSE

Apostolos Kritikos, George Kakarontzas, Ioannis Stamelos
Computer Science Department, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
akritiko@csd.auth.gr, gkakaron@teilar.gr, stamelos@csd.auth.gr

Keywords: Reuse, Free Libre / Open Source Software (FLOSS), Reuse process, Software components.

Abstract: It is clear that Free Libre / Open Source Software (FLOSS) has demonstrated increasing importance continually for some years now. As a result, millions of lines of code are becoming available online. In many cases, this code, is carefully designed, implemented, tested and therefore represents a very good option for reusability. Lately, more and more companies, especially Small and Medium Enterprises (SMEs), are reusing open source code to develop their own software. Source code forges such as SourceForge, Google Code etc., serve as component pools providing plenty of alternatives. In this work we are proposing a semi-automated reuse process model for discovering open source code online, based on the requirements of the system under design. This model illustrates the greedy approach of a reuse engineer, who wishes to reuse as much code as he can and implement the least possible.

1 INTRODUCTION

Code reuse is not a new phenomenon. Both software companies and individual developers know that there are certain blocks of code which form classic components in most of the commercial software projects. Moreover, there is the case where code that has been developed for specific requirements, serves as a base for a similar project that some future client requested. We usually refer to this kind of code as legacy code.

The vast adoption of FLOSS brought to surface the collaborative development of software. In addition the code of this software was made freely available online, allowing everyone to see, alter and in many cases even commercialize the derived work. This new development culture led to millions of free lines of code that transformed the WWW to a huge pool of reusable code which was lately organized to large code repositories that are known as forges (SourceForge, Google Code, etc.).

This paper is an experience report trying to capture specific, discrete steps the reuse engineer takes in order to reuse as much source code as possible. We then make an attempt to organize these steps in a semi-automated reuse process model. In this work we use the term *'reuse engineer'* to identify the *role* who attempts to reuse code by adapting either the code to the system under development or the system under development to the retrieved code, or both. The reuse engineer can be any developer especially in contexts where a systematic reuse program is absent, which is very often the case with SMEs, or it may be an actual engineer who has been assigned the task of retrieving and adapting reusable components in more systematic reuse approaches.

The rest of the paper is organized as follows. In Sec. 2 we propose a place for our model within the software product's life cycle. Sec. 3 describes case studies and the speculations arose by them which led to the process model. Sec. 4 provides a detailed description of our process model. Sec. 5 discusses related work. Finally the last section summarizes our conclusions and provides speculations for further research.

2 THE REUSE PROCESS INSIDE THE SOFTWARE PRODUCT'S LIFECYCLE

Software, as any other type of product has its lifecycle. In (ISO/IEC, 2002) the phases of a product's lifecycle are defined as follows: (1)

Concept (2) Development (3) Production (4) Utilization (5) Retirement.

Although the nature of the lifecycle of a software product might consist of slightly different phases, it is obvious that any attempt for code reuse will take place in the activity of software construction or the activities of extension / customization of the software product. These activities take place during the phases of development, production and utilization in the aforementioned product's lifecycle scheme.

In order for a software product to be able to benefit from code reuse, its initial description needs to be decomposed to small, simple, stand-alone requirements. Given that such a pre-process was made, each of the aforementioned requirements could represent a component to be implemented, or found from another source, and be reused after possible adaptation.

The component-based approach, as mentioned in (Crnkovic, 2006), is based in code reuse in the sense that existing components are combined in order to form the desired software. As far as the product's lifecycle in this approach is concerned, (Crnkovic, 2006) proposes a variation of the Waterfall model, which is called Component-based Waterfall model. This modified waterfall model, follows the same phases as the classical one, which are: (1) Requirements (2) Design (3) Implementation (4) Verification (5) Maintenance. The only difference is that in each one of this phases we work with components.

Both the abstract product lifecycle model and the Component-based Waterfall Software product lifecycle pinpoint the fact that code reuse, as a process, fits in the phases of code implementation or maintenance, where source code is being produced.

3 CASE STUDIES: SOFTWARE DEVELOPMENT BASED ON CODE REUSE

Reuse engineering is based in covering the requirements of the software product we are about to implement piece by piece. In order to be able to work this way we need to define the notion "piece of software". Most of the reusable code exists in open source software repositories. Additionally it is a common practice for open source software developers to organize their code in components, bigger or smaller.

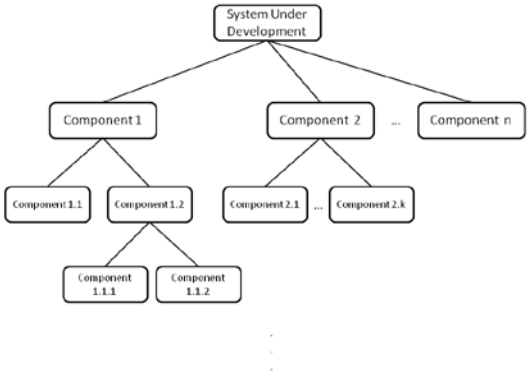

Figure 1. System under development decomposed in components (component tree)

With this in mind we can now go back to requirements and organize them to possible components following an approach similar to the one depicted in figure 1.

Initially, we consider each requirement as a separate component. Then, based on how complicated a function each one of these components encapsulates, we either decompose them to simpler, dividing their functionality to trivial ones, or not.

Eventually we will come with a tree structure that has as a root node the software product itself, and leaves, the components that need to be implemented in order to successfully implement the software product as a whole.

As long as we have this set of components at hand, we can start searching for their implementations in reusable code repositories. During this component "safari" we might face one of the following situations:

- The component we seek exists: In this case all we have to do is customize and integrate this component with the rest of our work.
- The component we seek does not exist, but subsets of it do: In this case we might need to go back to our component tree and extend it by breaking the component which we are currently dealing with, to simpler ones.
- The component we seek does not exist and dividing it to simpler seems more time consuming than actually implementing it: In this case there is no other option but implementing the component from scratch. Given the fact that in this case the component is usually a trivial artifact to implement we can always refer to development forums or online courses to "reuse" trivial snippets of code (for example read from or

write to a file, creating a Java Comparator, etc.). Usually a Google search does the trick.

In order to test the effectiveness of code reuse in action we experimented with two different scenarios which we describe here.

Searching for a log-in component. The objective is to seek for a reusable component that implements a web based login functionality in the form of a Java Bean.

The code reuse process is the following:

1. We are going for a quick solution (therefore we use Google) and search for "login java bean".
2. The second result is entitled "Authenticating users using a Java Bean". It looks promising.
3. We find source code available and a good documentation of what we are trying to do with this specific component in the web page we visited. Also personal information about the author informs us that he is a researcher and a developer in a software company.
4. There is no sign of copyright but there is no sign that the code is under any kind of open source license either.
5. Most probably, after a personal request to the author we will be able to reuse it.

During this process:

- We needed approximately 30" (seconds) to perform the search
- We needed approximately 5' (minutes) to have a first glance for the integrity of the site

In total: 5.5' minutes.

Wordnet Handler – Double code reuse to surpass library conflict: We have developed a Java class that serves as a handler for WordNet, the lexical database for the English language. In this implementation we use Java WordNet Library (JWNL) as a means to connect and handle WordNet. We want to embody our work to the bigger software product we are currently working to.

A major conflict with JWNL library, while trying to deploy our work as a Java OSGI bundle, forces as to use another Java compatible WordNet library. Logically, the handler's code will need to be rewritten as well.

We have spotted our new library candidate to be Java WordNet Interface (JWI). We would like to find reusable code to create a new WordNet handler class too. While searching to the documentation of the JWI library, at the official site of the library, we come across a sample class that implements most of the desired functionality. Instead of adapting the newly discovered reusable code we decide to try an experiment. To reuse the code we discovered to adapt our first handler implementation. The fact that WordNet provides specific data makes all library implementations similar and, as expected, their API's too. Combining this insight with the reusable code we have in hand, we come up with a new Java WordNet Handler in less than an hour. More specifically we needed:

- 10' to search for reusable code for JWI (the new library)
- 5' to become familiar with this code
- 30' to alter the old handler in order to use JWI
- 10' to test functionality

In total: 55' < 1 hour

For consistency reasons we mention that our initial implementation was also a product of code reuse. In order for our initial handler to come to its final version the timeframe, respectively, was:

- 1 ½ hours to search for reusable code
- 4 hours to customize and adapt the reusable code to the general needs of the project
- 2 hours to test our final code

In total: 7,5 hours.

The observations made during the above mentioned cases studies and other similar to them, lead us to the reuse process that we describe next.

4 A SEMI-AUTOMATED OPEN SOURCE SOFTWARE REUSE PROCESS

In this section we try to organize the knowledge derived from the case studies of the previous section to a more formal, schematic form. A model. Based on the aforementioned speculations we propose an open source software reuse process model (see figure 2). Although it might seems a bit daedal at first glance, once explained it is becomes really simple to understand and follow.

We start by defining the software product that needs to be developed (from now on we will refer to it as System Under Development). It can be considered as a unique component. Therefore, it is possible to be available in reusable code repositories. The reuse engineer performs a search to source code forges. If the search is successful, one or more results are returned. The reuse engineer proceeds then in code adaptation, packs the derived work and the software product is ready to be handed out to the customer. At this point one might notice that no specific methodology for choosing the best component (in case our search returns more than one) is being proposed. While this is true, it is not an omission. In this work we choose to introduce our model in a basic form, revealing its core

functionality. Component evaluation was intentionally left as an open issue for future research.

Once the reuse engineer has eliminated the possibility of finding reusable code for all the functionality he needs, he moves on by decomposing the System Under Development into components. This is the point where he, unintentionally most of the times, starts creating the tree of components we described in the previous section.

There is a small possibility that the System Under Development is too simple to be decomposed to discrete components. In this case our model proposes that it should be developed from scratch. Another possible scenario could be that the decomposition of the System Under Development and search for the derived components could require more time than the development of the project from scratch. Once the development from scratch decision is made, the System Under Development is being implemented, packed and it is ready to be handed out to the customer.

Most of the times, however, the requirements can be translated as discrete components. In this case, the reuse engineer must start searching for these components, one by one. In our process model this part of the development procedure is highlighted by the decision making rhombus entitled "UNIMPLEMENTED COMPONENTS?". Its role is binary. On one hand it starts the loop of trying to find reusable code for the unimplemented components. On the other hand it is the condition that ends the loop, and the whole process in essence, as it keeps track on whether there are any components left unimplemented. When no more functionality needs implementation, the System Under Development is considered finished, is being packed and is ready to be handed out to customer.

For every unimplemented component a sub process starts in order to decide whether reusable code can be found to implement the functionality needed or the source code of this component must be written from scratch.

As we mentioned in section three when breaking components to simpler ones, we face the danger to get lost in the procedure and eventually come up with having spent more time to find reusable code for a component than it would actually have taken us to implement it. In order to avoid this kind of pitfalls, our process model forces the reuse engineer, for each one of the components, to speculate on whether it is really worthy of implementing using reusable code. There are two possible scenarios where searching for reusable code should be discouraged:

- The component needed is very specific, therefore a lot of time might be spend in searching accompanied by a high probability of not returning any results.
- The implementation of the component is trivial; therefore it will take less time for an experienced programmer to implement it, than for the reuse engineer to spot reusable code that

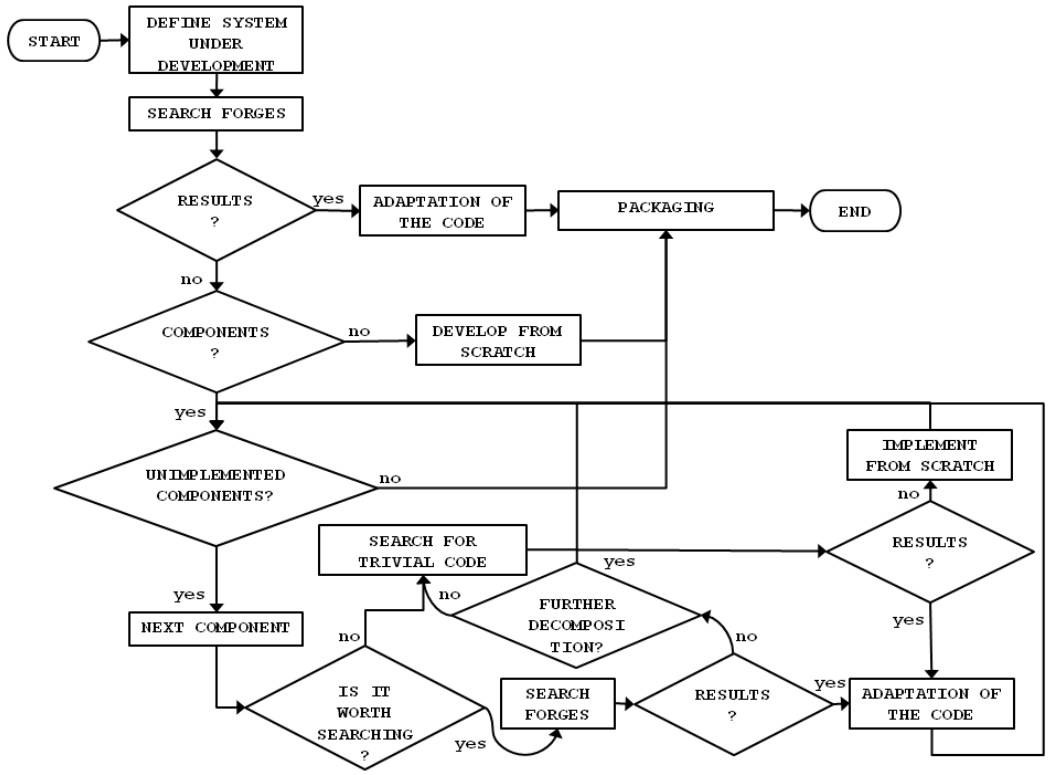

Figure 2. Open Source software reuse process model.

will cover its requirement.

If the reuse engineer decides that the component is not worth searching for reusable solutions he is left with only one option; implementing it from scratch. Looking at the model, though, one notices a different step to proceed than from scratch implementation: "SEARCH FOR TRIVIAL CODE". Developers were reusing code way before open source, or the various source code repositories flourish. This need for reuse comes from the empirical observation that some snippets of code are being found in most of software projects. Reading from or writing to a file, connecting to a database, creating a comparator in Java, are functionalities we meet so often, when it comes to software engineering, that have come to be considered *bibliography* code. It is this kind of code that we opt for the reuse engineer to find implemented by following the "SEARCH FOR TRIVIAL CODE PATH". This way, even when seeking for reusable components leads to a dead-end the reuse engineer can be sure that has exhausted every possible way of performing effective code reuse. After retrieving as many snippets of code as possible, the code is being adapted in the needs of the component under development, the code of the component is being finalized, and the reuse engineer is ready to move on to the next component. Of course, when all kinds of search fail, implementing from scratch is inevitable.

Finally we are going to examine the case where the component under development cannot be found as is in a repository of open source code and the reuse engineer needs to break it to simpler components of less functionality. No matter how complex a component he is dealing with, the reuse engineer needs to be sure that it does not exist in some code forge. Therefore, as our model illustrates, he performs a search to forges. After receiving no results he must examine whether the component can be further decomposed to simpler components. If not this means he deals with a component of medium or little complexity and therefore the whole process goes back to the previous paragraph where we discussed the role of the "SEARCH FOR TRIVIAL CODE" search / implementation model. In case the component can be decomposed to simpler ones our process model will consider them as "UNIMPLEMENTED COMPONENTS" and the reuse process will continue as normal.

We define the proposed model as semi-automated because, as it became clear by this section, the presence of the reuse engineer, is considered essential. When we speak about a reuse engineer, we refer to an expert, a software engineer trained to develop software using reusable code. As we already explained, normal developers can take on that role or in more systematic reuse approaches this role can be assigned to persons with this specific task as part of a development team.

5 RELATED WORK

Crnkovic et. al. present a modification of the waterfall process for component reuse (Crnkovic, 2006), in which there are two processes one for developing reusable components and another for developing systems with these reusable components. The authors discuss in detail the modifications of the activities of the waterfall model for system development with component reuse. To connect these two aforementioned processes they include an additional process called *'Component Assessment'* which should be carried out as much as possible independently from the system development to reduce time-to-market. In general an assessment activity should comprise the following: (a) Component discovery, (b) Component selection according its suitability for current and/or future products, (c) Component verification, and (d) Storage of the component and its metadata for future reference. Our proposed process can be used by reuse engineers to carry out the component discovery and selection in a more systematic way when they reuse FLOS software.

Some proposed processes for component retrieval from the Internet repositories aim at pushing the automation of this process as much as possible.

In (Hummel, 2007) a process called *Extreme Harvesting* is proposed, which uses unit tests that are developed in the context of an agile software development process (e.g. Extreme Programming) as a search criterion for reusable components. There are two variants of the Extreme Harvesting process, definitive harvesting and speculative harvesting. With speculative harvesting the reusable components retrieved are close to what the developers wanted but not a perfect match and the developers are required to adapt the system for the integration of the retrieved components. Therefore the process is not fully automated but is supported by an Eclipse plug-in.

Another tool supported process is proposed in (McCarey, 2005) in which the authors describe *Rascal* an intelligent agent, which oversees the development of new code and uses AI techniques to match the characteristics of the developed code with existing code from reuse repositories (e.g. the Sourceforge FLOSS repository). This process aims at more automation than (Hummel, 2007) since the search process is triggered by the intelligent agent

and the discovered components are presented to the reuser without his intervention. However, ultimately the developer is responsible for deciding the suitability of the retrieved components and for integrating them to the new system.

In relation to (Hummel, 2007) and (McCarey, 2005) our work aims at understanding the reuse process as a human activity first and then propose the tools for supporting this activity. Although tools such as the ones proposed in (Hummel, 2007) and (McCarey, 2005) are undoubtedly useful, our approach concentrates more at the moment on the reuse process itself, with the hope of better understanding the issues involved. We believe that a better understanding of the issues is also a prerequisite for more effective tool support.

Besides the searching and retrieval of reusable components, which is the basic area of our research, there is also a whole other spectrum of issues in software reuse in general and FLOS software reuse in particular. These include licensing issues and quality issues. There is progress towards supporting these aspects of reuse as well. For example the FOSSology project (Gobeille, 2008) is best known for finding the licensing of FLOS software which is a very important factor especially for commercial firms who wish to reuse open source software (Madanmohan, 2004). Projects such SQO-OSS (Samoladas, 2008) aim at providing quality related information for reusable software to enhance the trust of the users and re-users of FLOS software.

6 CONCLUSIONS AND FUTURE WORK

In this experience report we discussed about the role of code reuse when it comes to a software product's lifecycle and the software product's development process. We tried to provide, in the form of a case study the reuse engineer's approach in software development using concepts related to component based approach theory. Finally we proposed a semi-automated open source software reuse model in the form of a flow chart and presented how it organizes the steps, a reuse engineer is taking in order to create a software product with the less effort possible in terms of programming from scratch.

As we pointed earlier in this paper, this process model is a first attempt at providing a well defined way of implementing reuse engineering. Currently our model requires the presence of an expert, a reuse engineer, in order to take various kinds of decisions such as whether a component needs to break to simpler ones or not, which one of the reusable components discovered should we use to our implementation and why, what kind of adaptation the reusable code needs and so on and so forth.

As future research we would like to examine the possibilities of providing an even more automated process model that will be able to deal with some trivial although essential decisions such as the proposal of the best component in case the search returned more than one candidates based in specific metrics. Another interesting approach could be to try and measure the fitness of a component inside the system under development. By fitness we mean the similarity a component has with the others in terms of design patterns, coding style, quality metrics, etc. Once it reaches a certain level of maturity, the process model could ultimately be transformed into a tool using the open source forges as a reusable software pool providing a semi-automated way to any developer who wishes to discover and evaluate alternatives of free, reusable code.

REFERENCES

ISO/IEC 15288, System Engineering – System Life Cycle Processes, First Edition, ISO/IEC, 2002.

Crnkovic, I., Chaudron, M., and Larsson, S. 2006. Component-Based Development Process and Component Lifecycle. In *Proceedings of the international Conference on Software Engineering Advances* (October 29 - November 03, 2006). ICSEA. IEEE Computer Society, Washington, DC, 44. DOI= http://dx.doi.org/10.1109/ICSEA.2006.28

Oliver Hummel and Colin Atkinson: "Supporting Agile Reuse Through Extreme Harvesting", in proc. of the 8th International XP Conference, pp. 28-37, Springer, 2007

Frank McCarey, Mel Ó Cinnéide and Nicholas Kushmerick: "Rascal: A Recommender Agent for Agile Reuse", Artificial Intelligence Review, vol. 24, no. 3-4, pp. 253-276, Springer, November 2005

R. Gobeille: "The FOSSology project", In Proceedings of the 2008 international Working Conference on Mining Software Repositories (MSR '08), pp. 47-50, ACM, 2008

T.R. Madanmohan and R. De', "Open Source Reuse in Commercial Firms", IEEE Software, vol. 21, Dec. 2004, pp. 62-69

I. Samoladas, G. Gousios, D. Spinellis and I. Stamelos: "The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation", IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software, pp. 237-248, Springer, 2008