2010 Seventh International Conference on the Quality of Information and Communications Technology

Component Recycling for Agile Methods

George Kakarontzas
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece, and
Department of Computer Science and Telecom.
TEI of Larissa
41110 Larissa, Greece
Email: gkakaron@teilar.gr

Ioannis Stamelos
Department of Informatics
Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
Email: stamelos@csd.auth.gr

***Abstract*—Given the increasing size and complexity of today's systems, reusability is an important quality aspect. In this work we consider development and reuse of reusable components in the context of agile methods. To distinguish the proposed approach from the more established systematic reuse approaches we call our proposal *component recycling* instead of component reuse. For the development of recyclable components we show how inherent characteristics of agile methods, particularly the provision of useful and complete partial components during the lifecycle, can be used constructively for the development of components. These components are placed in a component repository for later recycling. The requirements implemented at each iteration serve as the design rationale for the components and distinguish earlier components from their later versions providing tracing.**

I. Introduction

Systematic reuse is achieved today with plan driven approaches such as the software product line approach [1]. These approaches require extensive planning so that assets are flexible enough to accommodate the needs of multiple products. On the other hand agile approaches aim at delivering high-quality software to the customers and they value this as their primary objective. Reuse in the context of agile approaches has received minor attention so far and the practices followed by most agile methods are not considering reuse practices at all. However we believe that reuse benefits such as increased quality, time performance and cost reduction are important for satisfying the main objective of agile methods which is the customers' satisfaction.

The rest of the paper tries to address reusability, or recycling as we call it, in the context of agile methods. In Sec. II, we discuss understandability metrics that are related to reuse and we examine the evolution of these metrics during the iterations of an agile process with an aim to improve them suggesting key practices for the reuse potential of the developed modules. Then in Sec. III, we elaborate on the key reuse practices that can be used in full-fledged agile processes to increase the potential for reuse and exemplify the practices with examples from a real case study. This 'merging' is similar to the approach of Agile Modeling [2] which adds modeling practices to agile processes such as XP. The result of our proposal is a process which is agile but reuse oriented. Next in Sec. IV, we describe related work. Finally in Sec. V, we provide future research directions and conclude.

II. Reuse improvement

Reusability is improved with understandability and understandability in general is improved with simplicity. The simpler the recyclable components the more understandable they will be. There are many examples of reuse assessment based on established complexity metrics and also new metrics developed specifically for reuse. To mention a few in [3] the reusability of class libraries in Java and Eifel is examined. The authors use a subset of the Chidamber and Kemerer metrics to evaluate reusability. In [4] the authors develop new coupling measures for assessing the reusability of Java components. In [5] the author suggests that a black-box view for component reusability is the most important, ignoring internal factors. The author then presents the most influential metrics for the reusability of classes and combines these metrics in a single formula (a new metric called "Reusabity for a class Rc") that can be used as a reusability oracle when developers consider including a class in a reuse repository. In this work we used the popular Chidamber and Kemerer metrics [6] as good indicators of understandability and we examined the evolution of these metrics for the classes that evolved from the first to the fourth and last iteration of an iteratively developed project. The selected project is used as teaching material for Object Oriented software construction in a known university and therefore good OO principles of design and implementation are followed throughout. We can safely assume that more complicated commercial projects will most certainly exhibit worst values for the CK metrics than those mentioned here.

We have used the open source *cjkm* tool [7] to collect the Chidamber and Kemerer metrics as well as two additional metrics which relate to complexity for ten classes which existed throughout the project's iterations. The metrics on the average increased during the iterations of the project as can be seen in Table I. In this table we have used a simple sum of the metrics' value as an indication of the complexity C in which all eight metrics are considered to contribute equally in the total complexity of classes (i.e. $C = \sum_{i=1}^{8} m_i$). The two

978-0-7695-4241-6/10 $26.00 © 2010 IEEE
DOI 10.1109/QUATIC.2010.72

additional metrics in Table I are the afferent coupling metric (Ca) and the number of public methods (NPM).

TABLE I
EVOLUTION OF UNDERSTANDABILITY METRICS THROUGH ITERATIONS

Iteration	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM	C
1st	4.1	0.9	0.2	2.1	9.6	0.9	1.9	2.6	22.3
2nd	7.3	0.9	0.3	3.5	17.8	11.4	3.4	5.7	50.3
3rd	9.3	0.9	0.3	4.0	21.7	15.4	3.7	6.6	61.9
4rth	11.3	0.9	0.5	4.1	26.7	22.7	4.1	7.6	77.9

Complexity increase during the iterations is *inevitable* because at each iteration more requirements are implemented and therefore the source code for these requirements will have an effect in the objects involved. We can control complexity following software engineering principles consistently (e.g. low coupling, high cohesion etc.), but regardless more requirements imply additional complexity. However one of the premises of agile methods is that each iteration delivers production quality software that partially addresses the functional and quality requirements of the application. This is important because it reduces the risk and also helps steering the requirements in the right direction since clients can use the functionality and gain a better understanding of the product. This observation combined with the fact that early iterations' objects are less complex suggests that is meaningful to keep these early snapshots of our application objects for future reuse, since (a) they provide a useful service by definition and (b) they are less complex than the respective components at the end of the project. In a sense this is what happens in all iterative methods anyway: each iteration results in objects that are reused in the next iteration. We just extend this to future applications as well.

The benefits for keeping the early components for future reuse include the following: (a) *Reduced application affinity*: The final application will implement a number of requirements $RS_1 = (R_1, R_2, ..., R_n)$. A reuser will probably need a subset of these requirements plus some additional ones, lets say $RS_2 = (R_1, R_2, R_3, R_k), R_k \notin RS_1$. The code in relation to the requirements that do not belong in RS_2 but belong in RS_1 represents *noise* that the developer will need to handle by first understanding and then removing the unwanted services. Then she will need to test the component and finally integrate it into the new application. Notice that removing unwanted services from the public interface in general does not work since the implementation of a service affects several methods. (b) *Improved Classification*: The classification of application components will be much easier if they are classified exactly after they are implemented, because at this point (after the integration testing) we are assured that the components work correctly and we know exactly why these components were built the way they were. In other words, both the requirements of the iteration and the design decisions are *recent*. Therefore it is very easy to relate the service providers (i.e. the components) to the exact requirements that they implement at the end of each iteration. (c) *Improved Integration*: The association of the components with the requirements that they implement eases integration of the recycled components to new applications. The reason is that the developers decide to reuse components when the only thing they know are the requirements of the services that they should provide (i.e. when they have not started yet the implementation of these components). If it is easy to find components based on the requirements then it becomes feasible to adapt the application's architecture locally around the recycled component in a way that enables the integration of the recycled component to the new system. Bosch has observed that "Practitioners have found that 'as-is' reuse seldom occurs and that reusable components generally need to be adapted to match the system requirements" [8], and Crnkovic et. al. that "a tradeoff between desired design and a possible design using the existing components must be analyzed" [9]. Adaptation therefore works both ways: we need to adapt the recycled components but also assess alternative designs to host them. Early requirements-based identification of recycled components allows the maximum flexibility for the integration activity. (d) *Improved Quality Assurance*: In the absence of independent certifying authorities for software components, developers will need to establish trust using more traditional quality assurance techniques. The most widely used QA technique is testing. Both unit and integration testing play an important role in agile methods. Since tests are built and run consistently they also evolve as code evolves. Associating snapshots of tests with the snapshots of components in the reuse repository becomes therefore a pragmatic approach in establishing the required trust for recyclable components. Reusers can run the same tests and tests can serve as executable specifications for the recyclable components aiding understandability even further. Associating requirements and tests with the reused components in the repository may be proved a far more effective way of documentation than traditional textual approaches and does not impose additional overhead for the agile developers.

For the aforementioned reasons, we propose that the intermediate components that are produced during the evolution of an application's lifecycle should be kept in a repository and be associated with the requirements and the tests that are produced at each iteration. Furthermore each component should be related in a hierarchy with the evolution of the same component to its next version. Thus a 'root' component will implement the services related to the first requirements that were implemented, the components at level 2 with the requirements implemented in the second iteration and so on. The hierarchical organization helps in discriminating components through their requirements and provides a learning path for reusers who can study the evolution of the component throughout the lifecycle. Furthermore the reuser will choose the component version that better matches his or her requirements whereas with the current approach to reuse repositories the only available component for reuse is the "final" version which may include unwanted functionality and dependencies and be much more difficult to understand than earlier versions.

The main classes for the repository are depicted in Fig. 1. Each `Application` belongs to an `ApplicationDomain` and is built in a number of `Iterations`. Each `Iteration`

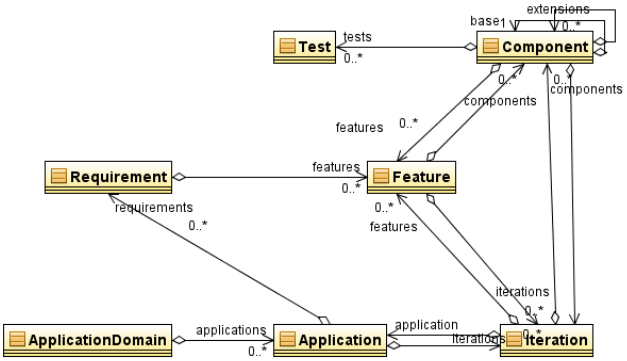

Fig. 1. Main classes of the IID recycling repository

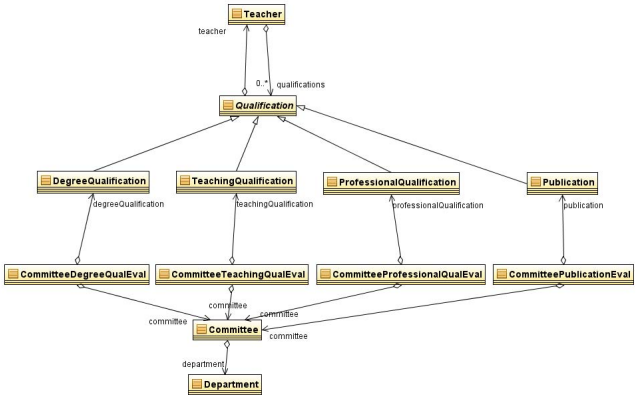

Fig. 2. A subset of the application's components

delivers a number of `Features` which are composing elements of high-level application `Requirements`. A `Component` participates in a number of `Features` and is built in an `Iteration`. Furthermore `Components` are associated with the unit `Tests` that verify their behavioral correctness. Each `Component` can have a base `Component` as its parent and a number of extension `Components` in the hierarchical organization of the repository.

Next we will describe the additional practices for the production of recyclable components in the context of a generic agile methodology.

III. Reuse Practices in the Context of an Agile Methodology

The practices that we propose for populating the reuse repository with components are explained in the next three sub-sections and they are: (a) A modified requirements prioritization practice, (b) component classification in the repository using unit tests and (c) reuse refactorings . To illustrate how these practices work, we will be using examples based on a real case study in which we applied the approach proposed in this work. The case study is a Java Enterprise Edition 5 (JEE5) component-based application which is used by an educational institution. The institution each year needs additional personnel to cover mostly laboratory courses but also theory courses for which the permanent staff is absent or unavailable. Applicants provide evidence for their qualifications including university degrees, professional experience certificates etc. An evaluation committee at each department evaluates the evidence for each applicant that applied for a teaching position and publishes a formal evaluation report which lists all applicants for a position in their evaluation order. The process for the evaluation is legislated and the rules for the evaluation are very specific and quantitative. The department committee decides if the evidence provided are real and if they are related to the position. Unrelated or unreal qualifications are not considered at all during the evaluation process. However each type of qualification is quantified. For example the years of professional experience are calculated using a formula that takes into account both the duration of the employment and the monthly salary. With the salary parameter it is determined if the employment was full-time or part-time, and if it was part-time only a percentage of the duration is attributed to the applicant.

The system consists of four modules: (a) *Applicant module*: This module allows an applicant to fill his or her qualifications online and apply for positions. The applicant prints the application from the system and submits the printed application as well as printed evidence for his qualifications in the secretariat office of each department. (b) *Evaluator module*: This module allows the evaluators to check the authenticity and specificity of each qualification in relation to a position. The system makes the required calculations based on the authentic and specific qualifications and produces the evaluation reports. (c) *Secretariat module*: This module allows the departments' secretaries to publish the available positions, accept applications and check them into the system. Only checked in applications are considered, and (d) *Administrator module*: This module allows the administrator of the system to setup departments, courses, user accounts etc.

In Fig. 2 we can see a subset of the application components which are relevant to the discussion at subsequent sections. Each `Teacher` has a number of `Qualifications` which can belong to certain subclasses of the `Qualification` superclass (e.g. `DegreeQualification`, `TeachingQualification` etc.). Each `Department` has an evaluation `Committee` which creates evaluation records for each one of the qualifications (e.g. a `CommitteeDegreeQualEval` for a `DegreeQualification` etc.). Notice that since the system was developed in JEE5 these components represent *entities* which are mapped in a database using the Java persistence technology.

A. Requirements prioritization

In our approach we apply the practice of requirements' prioritization using the importance of the requirements for the clients, the risk that they represent for the project, but also

the *reuse potential* of each requirement. Requirements with greater reuse potential should be implemented first as long as they are also important for the users and after the high-risk requirements have been addressed. Prioritizing requirements this way improves the probability that our components will need less rework by the reusers. The reason is that components that are implemented in the first iterations will not contain methods, dependencies etc. that the reusers will need to understand and isolate or remove entirely because they are irrelevant to their application needs. These methods and dependencies will of course be introduced at later iterations since they are important for the current application, but they will be based on possibly more reusable components that we have already kept in our repository which do not contain these possibly unwanted and application-specific elements. The reuser will then pick up the component that seems to be a better fit for his or her reuse needs without the need to handle the complexity introduced by unwanted functionality. To judge the requirements according to their reuse potential we need to have a rough idea on the kind of applications that may be developed in the future to which the components might be useful. This 'rough idea' does not represent a specified application set as in software product lines, but rather an application domain that these components may be useful. In our case study for example an obvious candidate domain that we may reuse many of the application's components would be to evaluate candidates for private sector companies. Contrary to our current application scenario for which the public educational institution is legislated by governmental rules the private sector scenario is based more on the intuition of the evaluators and hence some application requirements will be less reusable than others.

Requirements can be expressed in many ways. Some of the most popular include *use cases*[10], *user stories*[11] and *features*[12]. In our approach we prefer the usage of features as they are defined in Feature Driven Development because their scale is small enough to be implemented from a few hours or days to a maximum of two weeks and they highlight the involved components. A feature is "a small client-valued function expressed in the form: action result object, with the appropriate prepositions between the action, result and object"[12]. On the other hand use cases can be as large as whole business processes and user stories are written in index cards and are intended for further development. We do not insist on the usage of features as long as the scale is appropriate (not too small, not too large) and the involved domain objects are highlighted in the requirements. With our approach features should be classified in relation to their reuse potential as *highly reusable*, *moderately reusable* and *application specific*. Features which are highly reusable should be implemented first, followed by features which are moderately reusable and lastly by the application specific features.

To give an example of this practice we consider a subset of the requirements of the evaluation system. We concentrate here on the requirements that are related to the evaluation of professional experience given as features:

1) Confirm the acceptance of a professional experience
2) Provide justification for the rejection or alteration of duration of a professional experience
3) Calculate the duration of a professional experience
4) Alter the duration of a professional experience
5) Adjust the duration of a professional experience based on the monthly salary

The features 1–2 are highly reusable since in every evaluation system conceivable the evaluators will need to accept or reject a professional experience presented by the applicant (feature 1) and in the case that they reject it for some reason they should be able to provide their justification for doing so (feature 2). Features 3–4 are moderately reusable because they involve the duration of the experience as an evaluation criterion. Although duration can be an important evaluation criterion in most cases, in some cases it is the subjective judgment of the evaluator that matters more than the actual duration. Finally the 5th feature is very application specific and introduces a lot of complexity in the involved components. Basically it states that the system should be able to reduce automatically the duration of professional experience using the monthly salary as an indication of what is considered a full month. In order to implement this feature we will need to 'pollute' our general professional qualification component with attributes (salary received and base salary) and methods (setters and getters) specific to this feature which will not be needed in future reuse scenarios. Also methods that are needed in more reusable features (e.g. the calculateDuration method) will need to be modified (e.g. by adding the base salary parameter and the actual code that reduces the duration). This is an example of a feature that alters many methods which makes (as we already mentioned) simply hiding the unwanted services from the public interface infeasible in the general case. Furthermore the implementation of this feature will require changes to the user interface both for the applicants who will need to fill in their monthly salaries and the evaluators who will need to assess it. The salary related fields will also be stored in the database tables.

Our approach allows the reuser to choose among the possible versions of a component in the repository the less conflicting with his current reuse requirements. Suppose for example that in a new evaluation system the reuser needs also to evaluate professional experience records however instead of an automatic quantifiable process he needs to allow the evaluators to assign a grade (e.g. in the scale 1-5) to each professional experience record. With this requirement the most appropriate version to use would be the first since salaries and durations are irrelevant. Furthermore even for the maintainer of an application we see benefits with this approach since the component versions linked to the features that they implement provide a *learning path* from the relatively simple early components to the complete final versions. The maintainer is enabled to study the evolution of the components as they were implemented and therefore understand them more easily.

B. Component classification in the repository using unit tests

Our approach for the classification of components in the repository uses unit tests and Test-Driven Development to explicitly scope a component's functionality. The developer defines a set of tests T as the set of tests that the component should pass. After passing this set of tests and refactoring the code the component is thought as a complete new version of either an existing component in the repository or a new component. If the component is new (i.e. it was just created as a result of this iteration's features) it will spawn its own hierarchy in the repository. If the component on the other hand already exists there are two different cases:

The first case is that the component is built as part of the current application. Then the set of tests of the component so far should also pass for the new component and additional tests should be added to verify the new features in which the new version participates. The hierarchy of the components created as part of the same application is thus flat. If previous tests are violated then this signifies as usual a problem with either the understanding of the features required, errors in the component's code or errors in the testing code. In any case this situation should be resolved as usual and the component repository should be updated accordingly to capture the correct result of the evolution. For example in Fig. 3 components X_{v1}, X_{v2} and X_{v3A} are iterative successive versions of the same component. Each version participates in some additional features and adds some additional tests. However each new version does not violate any of the previous tests or cancels any of the previous features. That is $T_{v1} \subseteq T_{v2} \subseteq T_{v3A}$ and $F_{v1} \subseteq F_{v2} \subseteq F_{v3A}$.

The second case is the construction of a new component for a future application based on a reusable component which is part of the repository. In that case as we already said the component reuser can choose any version of the component for reuse and not only the final version. If the reuser reuses the component by adding functionality without disturbing the existing functionality then there is a chance that the new component will satisfy the tests and provide the features of the reused components in addition to its own new tests and features. In such a case the reuser can classify the newly created component as an additional extension of the existing component which he reused. This is depicted in Fig. 3 with the component X_{v3B}. The reuser created this component by reusing a previous version of the component X (version X_{v2} and not the final version X_{v3A}. If the newly created component still satisfies the tests of component X_{v2} and provides its features then it can be classified as an extension of the component X_{v2}. Notice that the component is incompatible with X_{v3A}. Also notice that in the above discussion the term 'extension' is used in its more traditional sense and not as the well-known OO technique.

C. Reuse refactorings

Refactoring is the alteration of code for improving its internal structure without affecting its functionality [13]. Refactoring is one of the most powerful techniques for the improvement of Object-Oriented software and is one of the standard practices applied with agile methods. Refactoring in the context of agile methods is strongly supported by the use of unit tests which ensure the behavioral preservation after the refactoring. We do not propose any new refactorings specifically designed for reuse, because many already proposed refactorings are geared toward the improvement of maintainability (e.g. extracting a superclass or extracting an interface) and simplicity of the source code. In fact it has been shown that refactoring as a general practice improves the reusability of the source code [14].

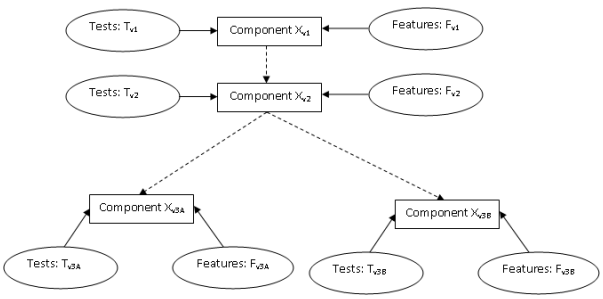

Fig. 3. Component hierarchies based on tests and features

The "reuse refactorings" practice is simply an advice to look for refactorings that are appropriate for the improvement of reusability of the source code for the foreseeable application domains that this code might be used. To give a simple example, the class `Teacher` in Fig. 2 has a very application-specific name, because a generic evaluation system will probably not be used in the context of an educational institution in which the applicants are teachers. This can be changed to the more general name `Applicant`. This could improve the chance that in the future and assuming we have created a large repository of thousands of components, a simple text search could yield the expected result. This renaming refactoring is in fact one of the simplest and is supported by the modern IDEs (e.g. the NetBeans IDE).

IV. Related Work

In [15] the authors propose a method called *Extreme Harvesting*. Extreme harvesting is a method for the efficient retrieval of reusable software components in the Internet. Searching is based on the use of tests which are developed in the context of an agile method (i.e. test matching). Programmers develop unit tests as usual in the context of an agile development process (e.g. Extreme Programming). However before developing the code that passes the already developed tests, they use the unit test as a search criterion in open source search engines such as Koders and Merobase.

In [16] we proposed the use of Test-Driven Development (TDD) practice of the agile methods for the creation of component variants of a software product line. We discussed how unit tests of an agile development method can be used for the organization of software components in a reuse repository. We found that many of the practices and principles of agile

methods (e.g. TDD, YAGNI) are in fact supportive towards the reuse of software components and their classification in a reuse repository.

In [17] the authors analyze the modification of Extreme Programming practices in a large project of the financial domain. Because of the large scale of this project many of the XP practices were modified and some additional practices were added. In relation to code reuse a particularly interesting modified practice is the so called *forward refactoring* practice. Refactoring is the modification of existing code, without modifying the provided functionality [13] for improved maintainability, understandability and other quality properties. In forward refactoring existing code is refactored so that it can be reused for the implementation of new functionality.

In [18] the authors propose the use of an intelligent agent, Rascal, which 'watches' the development of a class and proposes 'similar' classes which already exist in a reuse repository (e.g. the Sourceforge OSS repository). The advantage of Rascal is that the searching process does not begin by the programmer but takes place automatically in the backstage by the intelligent agent. For the similarity matching the authors use AI techniques. The approach aims at supporting reusable components' retrieval in the context of an agile method without additional overhead for the developers.

Lately there is an intense interest on the application of agile development practices in the context of a Software Product Line approach [1], with relevant publications (e.g. [19], [20]) and a special issue of the Journal of Systems and Software [21].

Concluding, the convergence of agile methods with reuse practices, are classified in three distinct categories: (a) Usage of agile practices in the context of an SPL approach (e.g. [19], [20], [21]), (b) Usage of isolated agile practices or modifications of these practices for the development of reusable components (i.e. development for reuse) (e.g. [16], [17]), and (c) Support for searching and retrieval of reusable components in the context of an agile method (i.e. development with reuse) (e.g. [15], [18]).

Our work aims at supporting reuse for companies that develop software using an agile approach. Instead of concentrating on an isolated practice we proposed a number of modified agile practices aiming at a holistic approach for both development with and for reuse in the context of an agile process.

V. Conclusions and future research directions

In this work we presented an approach to building a recyclable component repository that exploits the iterative development style of agile processes. We strongly believe that the final versions of the domain or business components of a software project are very application specific and hence they are rarely reused. However one can imagine an agile development project as a journey that starts from a rough general idea and ends with an application-specific set of components, with intermediate stations at each iteration's end. These intermediate stations, as we saw in Sec. II, are more *reusable grounds*. We proposed a set of practices that collectively may help the developers increase the reuse potential of their components and create a reuse repository.

Future work includes more case studies, a more usable component repository and experimental validation of the improvement of reuse using the proposed approach.

References

[1] Paul Clements and Linda Northrop: "Software Product Lines: Practices and Patterns", Addison-Wesley, 2002

[2] Scott Ambler: "Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process", Wiley, 2002

[3] S. Araban and A. Sajeev: "Reusability Analysis of Four Standard Object-Oriented Class Libraries", Software Engineering Research and Applications, Springer, pp. 171-186, 2006

[4] G. Gui and P. D. Scott: "Ranking reusability of software components using coupling metrics", Journal of Systems and Software, vol. 80, no. 9, Sep. 2007

[5] J. Barnard: "A new reusability metric for object-oriented software", Software Quality Journal, vol. 7, pp. 35-50, Mar. 1998

[6] Shyam R. Chidamber and Chris F. Kemerer: "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, IEEE, June 1994

[7] Diomidis Spinellis: "Tool Writing: A Forgotten Art?", IEEE Software, vol. 22, no. 4, pp. 9-11, IEEE, July/Aug. 2005

[8] Jan Bosch: "Superimposition: a component adaptation technique", Information and Software Technology, vol. 41, no. 5, pp. 257-273, Elsevier, Mar. 1999

[9] Ivica Crnkovic, Michel Chaudron, and Stig Larsson: "Component-Based Development Process and Component Lifecycle", International Conference on Software Engineering Advances (ICSEA'06), p. 44, IEEE, 2006

[10] Frank Armour and Granville Miller: "Advanced Use Case Modelling", Addison-Wesley, 2001

[11] Kent Beck and Cynthia Andres: "Extreme Programming Explained: Embrace Change, 2nd ed.", Addison Wesley Professional, 2004

[12] Stephen R. Palmer and John M. Felsing: "A Practical Guide to Feature-Driven Development", Prentice Hall, 2002

[13] Tom Mens and Tom Tourwe: "A Survey of Software Refactoring", IEEE Transactions on Software Engineering, vol. 30, no. 2, pp. 126–139, IEEE, February 2004

[14] Raimund Moser, Alberto Sillitti, Pekka Abrahamsson and Giancarlo Succi: "Does Refactoring Improve Reusability?", 9th International Conference on Software Reuse, LNCS vol. 4039, pp. 287–297, Springer, 2006

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

[16] George Kakarontzas, Ioannis Stamelos, and Panagiotis Katsaros: "Product Line Variability with Elastic Components and Test-Driven Development", International Conference on Inovations in Software Engineering (ISE'08), pp. 146–151, IEEE Computer Society, 2008

[17] Lan Cao, Kannan Mohan, Peng Xu and Balasubramaniam Ramesh: "How Extreme Does Extreme Programming Have to Be? Adapting XP Practices to Large-Scale Projects", 37th Annual Hawaii International Conference on System Sciences (HICSS'04)-Track 3, IEEE Computer Society, 2004

[18] 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

[19] Ralf Carbon, Mikael Lindvall, Dirk Muthig and Patricia Costa: "Integrating Product Line Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design", First International Workshop on Agile Product Line Engineering, 2006

[20] Kun Tian and Kendra Cooper: "Agile and Software Product Line Methods: Are They So Different?", 1st International Workshop on Agile Product Line Engineering (APLE'06), IEEE Computer Society, 2006.

[21] Kendra Cooper and Xavier Franch (eds.): "Agile Product Line Engineering", Journal of Systems and Software, vol. 81, no. 6, June 2008