

USING FEATURE MODELS FOR REUSABILITY IN AGILE METHODS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MARCIN JEDYK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

MAY 2011

Approval of the thesis:

USING FEATURE MODELS FOR REUSABILITY IN AGILE METHODS

submitted by **MARCIN JEDYK** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Ali Doğru _____
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Prof. Dr. Adnan Yazıcı _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Ali Doğru _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Ahmet Coşar _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Halit Oğuztüzün _____
Computer Engineering Dept., METU

Assist. Prof. Dr Reza Hassanpour _____
Computer Engineering Dept., Çankaya Univ.

Date: _____

ABSTRACT

USING FEATURE MODELS FOR REUSABILITY IN AGILE METHODS

Jedyk, Marcin

M.Sc., Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali Dođru

May 2011, 84 pages

The approach proposed in this thesis contributes to implementing source code reuse and re-engineering techniques for agile software development. This work includes an introduction to feature models and some of the Feature Oriented Software Development (FOSD) practices to achieve a lightweight way of retrieving source code. A Feature model created during the course of following FOSD practices serves as an additional layer of documentation which represents the problem space for the developed application. This thesis proposes linking source code with such a feature model for the purpose of identifying and retrieving code. This mechanism helps with accessing the code segment

corresponding to a feature with minimal effort, thus suits agile development methods.

At the moment, there is a gap between feature oriented approaches and agile methods. This thesis tries to close this gap between high-level approaches for software modelling (feature modelling) and agile methods for software development.

Keywords: Source code reuse, Feature Model, Agile methods

ÖZ

KIVRAK METOTLARDA YENİDEN KULLANILABİLİRLİK İÇİN NİTELİK MODELİMİ KULLANIMI

Jedyk, Marcin

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Ali Doğru

Mayıs 2011, 84 sayfa

Bu tezde önerilen yaklaşım kıvrak yazılım geliştirme için kaynak kod yeniden kullanımı ve yeniden yapılandırmasına katkıda bulunmaktadır. Bu çalışmada kaynak koduna hafif bir yöntem ile ulaşım için Nitelik Modelleri ve Nitelik Yönelimli Yazılım Geliştirme (Feature Oriented Software Development (FOSD)) kullanımı önerilmektedir. FOSD uygulaması ile oluşturulan nitelik modeli, geliştirilen uygulama için problem uzayının belgelenmesinde bir ek katman olarak da hizmet etmektedir. Bu tez, kodun tanımlanması ve ulaşılması için kaynak kodunun nitelik modeli ile ilişkilendirilmesini önermektedir. Bu mekanizma ile bir niteliğe karşı düşen kod parçasına minimal bir çaba ile ulaşma mümkün olmaktadır ve böylece kıvrak geliştirme metotlarına uygunluk

sağlanmaktadır.

Şu an, Nitelik Yönelimli yaklaşımlar ve kıvrak yöntemler arasında bir bağlantısızlık mevcuttur. Bu tezde yüksek düzeydeki yazılım geliştirme yaklaşımları (nitelik modellemesi) ile kıvrak yazılım geliştirme metotları arasındaki bu boşluk doldurulmaya çalışılmaktadır.

Anahtar Kelimeler: Kaynak kodun yeniden kullanılabilirliği, Ana özellik modeli, Kıvrak metotlarda

To Maryam

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Assoc. Prof. Dr. Ali Doğru who was always ready to share his wisdom and insightful ideas while I was working on my thesis. His knowledge on variety of topics and disciplines significantly contributed to this work. I appreciate his help and motivation while I was studying in Middle East Technical University.

I would also like to thank Assoc. Prof. Dr. Tolga Can for his help and support, both, when I was an Erasmus student at METU and when I became a regular student. His help and encouragement motivated me to apply for Masters programme at METU.

I would also like to express my gratitude to my parents who supported me during studies and motivated me to work hard and achieve my goals. They were always very supportive.

My friends, Maryam, Kemal, Masoud, Saeid and Eldi also helped me while I was working on thesis and I would like to thank them for their support and contribution.

At last but not least, I would like to thank my previous supervisor and Erasmus coordinator from Faculty of Computer Science and Management, Wroclaw

University of Technology, Dr. Jan Kwiatkowski. I would like to thank for his help while I was working on my first thesis while I was studying at METU.

TABLE OF CONTENTS

ABSTRACT	iv
ÖZ	vi
ACKNOWLEDGEMENTS	ix
TABLE OF CONTENTS	xi
LIST OF TABLES.....	xii
LIST OF FIGURES.....	xiii
CHAPTERS	
1. INTRODUCTION	1
1.1 Motivation	3
1.2 Thesis organisation	4
2. BACKGROUND INFORMATION AND CURRENT WORKS	6
2.1 Source code analysis.....	7
2.1.1 Source code re-engineering.....	8
2.2 Software development methodologies	11
2.2.1 Agile software development	15
2.3 Documenting software design	17
2.3.1 Domain engineering.....	18
2.4 Software reuse	23
2.4.1 Software product lines	25
2.4.2 Component-Based Software Engineering	27
2.5 Feature modelling.....	29
2.5.1 Feature trees and feature models.....	30

2.5.2	Feature oriented domain analysis (FODA)	32
2.5.3	Feature-Oriented Software Development (FOSD)	33
3.	USING FEATURE MODELS FOR REUSABILITY IN AGILE METHODS.....	35
3.1	Overview of the approach	36
3.2	Constraints and limitations of current work	45
3.3	Suggested usage of the approach.....	46
3.3.1	Roles and responsibilities.....	47
3.3.2	Code and feature models sharing and maintenance	48
3.3.3	Feasibility of the approach.....	50
3.4	Design of the system	55
3.4.1	Code analysis	56
3.4.2	Representing Features and Functions	58
3.4.3	Code retrieval.....	59
3.4.4	User interface and code visualisation.....	63
3.4.5	Creating sample project	66
4.	EVALUATION OF THE APPROACH	71
4.1	Test design and testing approach.....	72
4.1.1	Target group and environment.....	74
4.1.2	Test results and participants' feedback on the tool and approach.....	75
5.	CONCLUSIONS AND FUTURE WORK	77
	BIBLIOGRAPHY	82

LIST OF TABLES

TABLES

Table 4.1: Time required for retrieval and identification of a source code for particular feature. Measured for different participants with and without provided tool.....	75
--	----

LIST OF FIGURES

FIGURES

Figure 2.1 RUP phases and disciplines	14
Figure 3.1: Overview of creating project with proposed methodology	40
Figure 3.2: Process of identifying assets and source code reuse.....	41
Figure 3.3: Roles, assets and project allocation and access for proposed	50
Figure 3.4: Simplified model of function representationed model of function representation	60
Figure 3.5: Sample Java function - subject to source code retrieval and analysis, 1	61
Figure 3.6: Sample Java function - subject to source code retrieval and analysis, 2.....	61
Figure 3.7: Sample Java function - subject to source code retrieval and analysis, 3.....	61
Figure 3.8: Main layout of the interface.....	64
Figure 3.9: Additional view with listed (discovered) features	64
Figure 3.10: Functions call hierarchy for particular feature.....	65

Figure 3.11: Interface with function call hierarchy and discovered code for particularfunction	66
Figure 3.12: Creating new project	67
Figure 3.13: Interface for adding features to feature model.....	68
Figure 3.14: Sample feature model	68
Figure 3.15: @Feature annotated code in IDE.....	69
Figure 3.16: Function call tree with selected function to display in source code pane	70
Figure 3.17: Interface with selected function in function call tree and displayed source code	70
Figure 4.1: Sample feature model	72

CHAPTER 1

INTRODUCTION

Advancements in a field of a software engineering and computer science enabled use of software products in various fields such as automotive, defence, enterprise operations support, entertainment and many more. The growing importance and ubiquity of software projects influences the way in which software is designed, developed and maintained. Growing importance makes the software market bigger which implies growing competition on that market. This increases need for cheaper, more reliable and faster delivery of products to the end users.

Lowering costs and/or increasing quality of products are aims for many industries because it leads to growth of sales and profits. Different industries would have different approaches for achieving mentioned goals. For production industry lowering cost could be attained by decreasing materials waste without lowering quality of production and without increasing costs of labour above what was saved. In software production industry approach should be obviously different since different kind of 'material' is used. The obvious need for making software faster, cheaper and more reliable is not a new idea and up to now, lots of work has been done on software reuse field. Such works should lead to establishing new standards and making software development more a predictable discipline rather than art.

For lowering costs of software development, new approach like agile software development could be used which allows for developing software products with lower overhead. On the other hand, there are Feature-Oriented Software Development (FOSD) and Feature Modelling practices which allow for rapid modelling of the applications. Each of them is important and efficient approaches used in the software industry.

Someone could ask, why should we write again a source code which has already been written ‘somewhere by someone’? Until now, ‘reinventing the wheel’ is unfortunately a common practice in software industry. It is because software reuse practices are not very well established and in most of the cases it is simpler and easier for a company to write a code from a scratch rather than use once written code or even use market-available component. Of course, nowadays it is much better in terms of using available libraries, components, frameworks or execution environments (i.e. application servers) than in ’80 or early ’90. Nowadays developers have wide range of libraries (both free and commercial) as well as very powerful Java programming language which has build in many functions and data structures – now, when using Java programming language developers do not have to implement by themselves such structures as hash set, linked list, blocking queue, etc. Also, some very popular algorithms such as quick sort and very basic design patter – observer, are available for Java programmers without need of implementing them. Of course, even back in ’80 or early ’90 such data structures or algorithms could be included via libraries into programs, but there was a chance of data incompatibility when third part libraries were used. When Java-like widespread of data structures and basic algorithms is taken into consideration, it could be concluded that with advance of computer science, some higher level solutions or components will be also easily available for developers without need of fully implementing them – without need for

‘reinventing the wheel’. However, it is possible that in order to achieve that, new programming languages, paradigms or just new way of thinking about how software have to emerge.

Unfortunately it could take many years until software reuse will be advanced enough to use for example ‘online login form’ as easy as hash set in Java. Until it happens it would be worth to focus on more realistic software reuse techniques which also would increase quality, productivity and decrease development costs. It would be even better if a company could create its own reusable assets while working on their projects – such assets could be later used in other projects or even sold on a market to other companies generating additional revenue. The problem with current techniques for software reuse is they are very complex and expensive to introduce which is a serious barrier for many potential adopters.

1.1 Motivation

At the time of writing, there is no work which combines usage of agile development methods and feature models for rapid modelling the application and reusability. Lack of such approaches creates a niche for it. An approach which combines agile development with feature models for reusability should have low entry and exit barriers which mean that development teams could start using this kind of approach without need of high expenses and at the same time could abandon it without any obstacles. What is also important such approach should be easily comprehensible and should not add high overhead to standard development processes. Those who are familiar with agile development could come to a conclusion that such ‘lightweight’ and ‘developer friendly’ approach could be build upon similar ideas as agile development methods – without unnecessary bureaucracy and heavy processes.

The purpose of this work is to deliver an approach which meets above criteria - especially it should be close to agile development methods. Proposed approach will allow for using feature models for improving reusability with agile methods. Such approach will allow for reuse once written code, or rather 'retrieve once written code' basing on a feature model. In such case, source code would be a low-level resource (asset) and feature model would be a high level abstraction of it. Usage of such approach would allow developers for easy access to their or other programmers' source code which could have positive influence on an overall quality of developed products.

Another purpose of this work is to show that it is possible to implement and introduce such agile-like approach for source code reuse and re-engineering. By proposing such easy and lightweight approach for a complex task as software reuse, I want to provoke new way of thinking about software reuse – I want to encourage other researchers to think about other simple and easy to use approaches for other difficult problems in a field of software engineering. By changing way of thinking, we can solve complex problems in much simpler way – instead of providing complex solution for a complex problem, it would be better to make that problem simpler and provide simple solution. Proposed approach is not a silver bullet and at current stage, reusing source code would not be as easy as using a hash set in Java, however, reusing company's assets would become possible at low cost and without big overhead.

1.2 Thesis organisation

This thesis is organised as follows. Chapter 2 presents background information on source code analysis methods and source code reuse, features oriented approaches, software product lines and software development methodologies. That chapter focuses on various trends including those which are firm and

widely accepted by academicians as well as those which are relatively new and are object of evaluations. Understanding of basic concepts presented in that chapter is essential to understand and analyse proposed approach. Chapter 3 presents proposed solution with examples how does it work and how could it be used. Chapter 4 is an evaluation of proposed approach and describes how it was evaluated. That chapter allows us to understand how developers could use the approach and what their individual attitude towards it is. Chapter 5 concludes presented solution and points out possible further works and extensions to given approach.

CHAPTER 2

BACKGROUND INFORMATION AND CURRENT WORKS

This chapter presents current and past researches on various aspects of reusable software development and software development methodologies with focus on agile development. The core issues of this chapter are source code analysis and source code reuse, domain engineering, software product lines, feature modelling and software development methods. This chapter also sheds a light on feature trees and feature modelling. This chapter also focuses on various aspects of development methodologies – when proposed methodology is meant to be close to agile development, it is also worth to know principles of heavy development methodologies. It is important to understand how those methodologies affect development teams so proposing own lightweight methodology for source code reuse will be easier basing on that knowledge.

Software reuse issues and software development methodologies are fundamental for this thesis, thus related issues have been exhaustively described. Field of *software reuse* is very attractive from economical point of view, thus much attention has been paid to it in recent years. Agile software development methods seem to be well established and mature approaches for developing software products. However, issues of incorporating software reuse or source code reuse practices with methodologies which are ‘agile compatible’ have not

been investigated up to date. This chapter focuses on most significant and universal works in those fields and serves as an introduction to both topics.

2.1 Source code analysis

Source code analysis is a very wide concept and is being applied on variety of fields. Few examples of applicability of source code analysis are [6]: debugging, quality assessment (code metrics), fault location, comprehension or validation. Each programming language follows set of rules which is programming language grammar – those rules describe what constitutes syntactically correct source code - syntax. Besides syntax, semantic aspects of programming languages are also playing an important role in a construction of a language. Automated or even semi-automated source code analysis could give big advantages to software developers because such analysis makes it possible to analyse a code in a way that a single developer would not be able to do in a reasonable time (in particular cases). As Dr. David Binkley points out in his paper [6], potential benefits of using tools for source code analysis are very promising:

“The tremendous increase in the amount of software in use each year produces a growing demand for programmers and programmer productivity. Hiring additional programmers is costly and ineffective if the system under consideration cannot be broken down into pieces. Given the complexity of modern software, a more viable solution is tool support. Of growing interest are tools based on source-code analysis. Such tools provide information to programmers that can be used to coordinate their efforts and improve their overall productivity.”

Source code analysis could provide valuable, higher level information for developers of analysed code. The fact that information is presented at *higher level of abstraction* means it is more comprehensible for the developer.

Source code analysis is a multiple stage process which requires [6]:

- **Parsing**

This stage of source code analysis obviously requires usage of parser prepared for particular programming language (particular grammar). Writing a parser is not a trivial task, and there are parser generators available which can generate parser for particular grammar basing on user defined grammar description.

- **Preparing internal representation of the source code**

- **Analysis of such information**

Each process could be conducted by a separate tool which output is passed to another stage of analysis for further processing. For example, during source code parsing we can obtain such representation of the code, which will allow for preparing internal representation of the source code.

2.1.1 Source code re-engineering

Idea of reusing existing software, in particular – source code of applications is not new and some researches on that field have been done. Authors of the work [7] are focusing on two essential questions towards software reuse and re-engineering:

“1) What are the engineering solutions to be looked for in the existing software?”

2) What are the inter-component standards and component templates to use for recording them?”

It could be concluded that one very important aspect of software reuse and re-engineering is to provide an abstraction of written source code or components. With such abstraction it would be possible to apply reusable asset for different purposes. It does not necessarily mean that with such approach we could

develop a new system by simply connecting and combining different parts of source code – like in component oriented development. However, with having such source code described by higher level of abstraction it should be easier and cheaper to apply existing solutions to new projects and at the same time, reducing number of lines of source code that has to be written.

Authors of the work [7] proposed a five stage process for creating a portfolio of reusable components. Those stages are:

- **Candidature:**

This stage involves usage of various activities of source code analysis and as an outcome sets of modules are produced. Those sets are potential candidates which after proper decoupling and generalisation could be turned into reusable modules.

- **Election:**

During this stage, sets of modules produced during *candidature* phase are grouped into reusable modules.

- **Qualification:**

Outputs of this stage are functional and interface specifications of reusable modules produced during *election* phase.

- **Classification and storage:**

At this stage both reusable modules and theirs specifications are grouped into related category and taxonomy. In this way, reusable assets could be identified and discovered during reuse and re-engineering phase. Also, during this stage, a storage and repository for the reusable modules are set up.

- **Search and display:**

An output of this stage is a front-end interface for end users who will interact with the repository and will be benefiting from access to the

reusable modules. It is essential to make the interface and search procedures as simple and intuitive as possible in order to make work with reusable modules as easy and efficient as possible.

“Setting up a reuse re-engineering process may entail using a large number of methodologies and tools from different fields, mainly the reverse-engineering and re-engineering fields, as well as defining a new methodologies and creating new tools. Because most of the methodologies and tools provide only partial solutions the definition of a reuse re-engineering process must be founded on a reference paradigm”[7]

Proposed approach towards software re-engineering [7] seems to be complex and requires heavily involvement of developers or trained analysts who can conduct the processes. Such complexity and possible costs of creating portfolio could be a barrier for potential users of the methodology.

Except methodology described above [7] there are also other approaches towards source code reuse and reverse engineering. There are known approaches such *program slicing* and *concept assignment*. Both of them take different approach for extracting part of the code and as it will be later shown, both approaches have been combined to extract code more efficiently.

Program slicing [30] is isolation of program behaviour and reducing source code for each “behaviour” so it still will be valid and executable code but without redundant lines of code. In this case, redundant lines of code are such lines which are necessary for other *slices* but not the one we are extracting at particular time. In this way, source code could be ‘sliced’ and different functions or *subprogram* could be extracted. Programs are sliced based on *slicing criteria*s which are defining what should be extracted.

Concept assignment problem is “the problem of discovering individual human oriented concepts and assigning them to their implementation oriented counterparts for a given program” [5]. Such process is aimed to recognise concepts within a source code and assigning such concepts to various parts of the code and it requires understanding of it. There are three different approaches for concept assignment, and various tools could be delivered for each of them [5, 15]:

1. The code and generally, is “highly domain specific, model driven, rule-based” [15]. Systems aimed to tackle that problem are dependent on pre-populated databases which describe the domain which is analysed.
2. “Plan driven, algorithmic program understanders or recogniser” [5]
3. “Model driven, plausible reasoning systems” [5]

Each of the approaches has different characteristics and is aimed for different program sizes and are characterised by different precision. Solution 1 and 2 are efficient for small scale-programs while solution 3 can handle large-scale programs more efficiently. The drawback of approach 3 is its lack of precision and possibility of getting not precise results [5, 15].

2.2 Software development methodologies

Building complex software applications is a process which requires significant amount of resources, such as experienced personnel, time, support tools and knowledge. Such a complex task requires proper processes in order to support software construction. Because of that, software development methodologies have emerged to facilitate development lifecycle. The idea behind such methodologies is to develop a software product in defined and methodological way which should assure predictability of outcome and help participants to

understand what and how they should work. Following an established software development methodology allows to focus on “what” rather than focussing on “how” to do that. Another benefit of using a methodology for development purpose is that many people before has used particular approach and we could benefit from their experience and insight on usage the methodology.

There are many different approaches and methodologies for developing software products and they are varying between each other in several ways. Because of differences, some of them are preferred for particular types of projects which require specific features of particular methodology. Sometimes choose of methodology simply belongs to experience of a team, preference of project manager or culture of an organisation. It is never an easy task to change a methodology used within organisation because change of methodology also requires change of practices, tools and sometimes key people responsible for developing and managing software products.

One of the first widely used and recognised software design process *waterfall* cited for a first time in 1970. Waterfall is a design process composed of sequence of steps: requirements specification, design, implementation, verification and maintenance. That was industry-like approach where requirements and approach had to be up-front planned and agreed – without possibility to change anything during implementation stage. The fail-rate for that approach was high and probability of failure was higher for bigger projects. However, waterfall approach *could work* in a case of extremely straightforward and small systems where change or requirements or possibility of not predicting something are relatively low. That is of course more hypothetical because uncertainty and need for change during development are very common in software projects.

Over the time a concept of *iterative and incremental development* (IID) has emerged as a response for gaps of waterfall process. The main idea behind IID is to repeat activities like planning, requirements gathering, analysis & design, implementation, testing and evaluation in a cyclic manner during project development. The value which IID brings and waterfall lacks is feedback from users between iterations. In waterfall everything had to be set and agreed in advance – with zero flexibility and adaptability while IID benefits from feedbacks from stakeholders between iterations. Many modern development methods are based on IID and they vary between each other in few aspects. In context of IID it is worth to mention about Rational Unified Process (RUP) and agile methods because they follow similar philosophy but they vary in few details which makes big gap between them in terms of development culture, quality and success rate. It is important to compare RUP with agile methods because they have same roots, but few features made a huge difference between them.

RUP is an IID based methodology that is based on four iterations (phases). Those phases are:

- Inception phase
- Elaboration phase
- Construction phase
- Transition phase

Each of those phases has one main objective to be accomplished and at the end of each phase it is verified if such objective has been meet. In case of big projects each phase could last as long as few months. This is very long period and feedback and objectives verification could be delayed and in consequence, this methodology does not benefit from frequent feedbacks from product owners.

Tasks conducted within mentioned phases fall into one of six different engineering disciplines. Those disciplines are:

- Business modelling discipline
- Requirements discipline
- Analysis and design discipline
- Implementation discipline
- Test discipline
- Deployment discipline

RUP phases and disciplines are presented as on Figure 2.1.

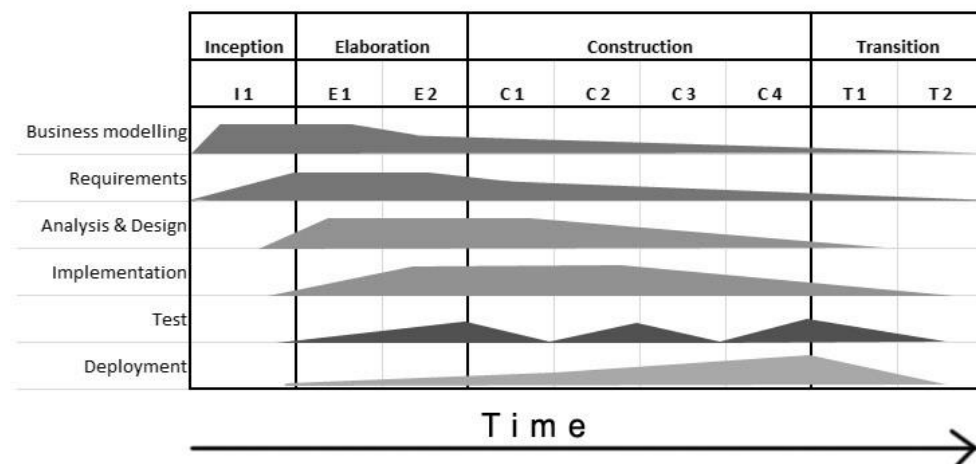


Figure 2.1: RUP phases and disciplines

Figure 2.1 reflects how disciplines are spanned across different phases and during each phase particular discipline could be more or less exercised. For example, it could be noticed that business modelling discipline is has big share of all tasks in first phase while tasks of deployment discipline have the biggest share of the tasks in last phase – transition.

RUP is considered as a heavy and document oriented software development methodology. It is designed for rather big projects for which using so complex and heavy methodology would have any sense.

Because RUP does not fit all sizes of projects – it is rather for big and complex one, alternative IID based methodologies have been proposed. Those alternatives are agile methodologies such as Scrum, XP, TDD and others which could be distinguished from RUP by various factors, for example length of iteration – ranging from one to six weeks [22]. Length of iteration has impact on projects and as Craig Larman states [22]:

“A three-month or six-month timeboxed “iteration” is extraordinary long and usually misses the point and value; research shows that shorter steps have lower complexity and risk, better feedback, and higher productivity and success rates. That said, there are extreme cases of projects with hundreds of developers where a three-month iteration is useful because of the overhead.”

2.2.1 Agile software development

Not everyone was satisfied with heavy, document-oriented and very bureaucratic methods of software development. As a response for those heavy methodologies, many lightweight methods have been proposed. Agile development methodologies are aimed to be as lightweight as possible. In such methodologies, a usage of as simple and easy to setup solutions could be observed. For example, instead of relying on heavy and complex tools such as Rational Rose developers are tending to manage development processes with help of whiteboards, post-it cards attached to walls and assigned tasks visible on a wall in a development office. Such approach guarantees that a simple and efficient environment could be easily implemented and at the same time, developers can see what tasks are currently assigned just by approaching the

wall. Such solutions have their roots in Toyota car production lines where visualisation and simplicity are keys for efficiency and 'lean production'.

Examples of those methodologies are Scrum, Extreme Programming (XP), Adaptive Software Development (ASD), Test-Driven Development (TDD) and Feature-Driven Development (FDD). Those methodologies have emerged before “Agile” principles have been founded and agreed on by a group of respected and famous software engineers. On February 2001 [17] a group of 17 software engineers has agreed on basic principles of Agile Software Development. As a result famous *Manifesto for Agile Software Development* has been written and signed by all 17 participants. The most essential values of that manifesto are:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following plan

The manifesto could be easily misunderstood and could lead to false conclusion that its authors do not value “processes and tools” or “comprehensive documentation” is not important. That is far from true.

The manifesto states that “individuals and interactions” are more important than “processes and tools” but it does not reject value of proper tools and processes. For agile methodologists, each member of a team is unique and his/her skills should be used in a proper way in order to drive project further. Following that, processes should be compatible with teams and its members, not vice versa.

The same is true for second principle – it is not matter how good software’s documentation is if the software itself is not working. However, working

software will not have full value for its customer if documentation is not delivered. Without documentation, usage, maintenance or further development could be impossible, thus there should be agreement between client and developers how detailed should be that documentation.

2.3 Documenting software design

Creating software projects requires involvement of different people who are interested in project completion and have their duties and responsibilities related to software development or software operation. Those individuals are called stakeholders. Stakeholders are an intrinsic part of every project of a significant size and play important role in project establishment and creation. They could contribute to the project development in various ways and usually have different perspectives on some parts of developed systems and different ideas how those parts should function. Each of them can contribute to project development in his/her way by providing insight on different aspects of the system. Because of that, their needs and concerns about developed system should be taken into consideration at every stage. Each stakeholder play different role during project lifecycle, have different objectives, different needs and different way of looking at the project. Moreover, some of them will participate only in specific stages of the venture or focus only on a small piece of the whole.

Depending on project, different stakeholders could be participating in system design and development. Example stakeholders are presented below:

- Customer
- Project manager
- System architect
- Programmers

- System integrators
- Testers

In order to efficiently use knowledge and experience of various stakeholders, good communication practices should be established. Such communication could be understood in terms of good project documentation (system representation) – which does not mean a complex and very detailed documentation. Most of the time different stakeholders would require different kind of documentation presenting different perspectives. Such documentation could be delivered in form of *views*, where different view reflects different level of abstraction and details of designed system. As Remco M. Dijkman et al. write: “Stakeholders focuses on a part of the design, which we call the view of that stakeholder (...) we way that a stakeholder focuses on certain design concerns and considers these concerns at a certain level of abstraction” [11].

Ensuring that stakeholders are well informed about each other goals and perspectives and that they could express and understand the system in their own way, could contribute to developing of high quality system. It could be very challenging and difficult to come with appropriate solution for documenting and modelling stakeholders’ perspectives, but once it is done, it would be a big contribution to software engineering approaches of developing systems.

2.3.1 Domain engineering

In terms of software engineering, a domain could be understood as knowledge or activities representing particular group of related systems. A characteristic of a domain is a shared vocabulary used by different stakeholders involved in activities within that knowledge area. There are various definitions and descriptions of domains and for example Mili et al state that domain is “defined by the common managed features that satisfy specific market or mission” [25].

Domain engineering could be defined as set of activities that lead to development of domain model used to create a family of related products. In other words, domain engineering is a process that leads to development of *reusable artefacts* (core artefacts) [16]. Because scope of domain could be used for developing multiple products for serving a particular market or mission, domain modelling is especially useful for creating reusable software, and serves as a base for developing multiple products, also known as *software product lines*. It is also defined as:

“a key process needed for the systematic design of an architecture and set of reusable assets (...) It is a systematic process that incorporates business criteria and that enable better decision to be made, reordered and revisited for further revision, and for process improvement based on learning”[14]

Another definition of domain engineering defined by Kang et al in early nineties is following [19]:

“[Domain engineering is] an encompassing process which includes domain analysis and the subsequent construction of components, methods, and tools that address the problems of system/subsystem development through the application of the domain analysis products.”

Going further, it could be stated that domain engineering activities lead to identification and generation of assets which could be used to generate profit of an organisation during multiple projects. Domain engineering and reusable assets creation is not solely about creating assets in-house, “producing reusable assets can be interpreted to include acquisition of assets developed outside the corporation, such as commercial off-the-shelf (COTS) products” [25]. Most likely after completing domain engineering steps variety of assets will be derived. Those assets could be:

- Domain scope

- Domain model
- Standardised development activates and processes
- Generic software architecture for the domain

Domain engineering is a multiple steps process and the following stages could be specified [14, 25]:

- Domain analysis
 - Domain identification and scoping
 - Making decision if it is worth to create domain for reuse
 - Selection and analysis of examples, needs and trends
 - Identification and cataloguing of commonalities and clustering features
 - Specifying reusable assets and classification for particular purposes
- Asset/component engineering
 - Develop reuse infrastructure, domain model and architecture
 - Representation of usable commonality and variability
 - Exploitation of commonality and variability
 - (Optionally) Assets acquisition
 - Implementation, certification, and packaging of reusable components

Of mentioned steps in domain engineering, domain analysis activities are crucial because at this stage, a decision will be made if it is worth to create infrastructure for reusable development of specific domain or not. This step will also determine shape of a domain and determine which features will be included for further development. Any mistake at this stage will be amplified and have big impact on further stages.

Developing a domain is more difficult than developing specific application because of several reasons. First of all, domain is developed to serve as a foundation for a family of more or less related products. This implies that domain model must be more generic and express higher level of abstraction than an application model. Also, when it comes to domain analysis, more examples and aspects should be analysed and taken into consideration to generate possibly universal domain specification. It is important that domain specification or domain model consists of only common and generic features that are suitable for variety of products within that domain. That is, domain model should describe family of related products in terms of commonality and variability [18]. This requires good knowledge base, experience and insight into a particular domain from engineers responsible of domain analysis and design. It is also important to mention that in order to create a good domain model, requirements engineering process should be suitable for delivering reusable domain models. Besides studying exemplary products, domain modeller or requirements engineer should also focus his attention on gathering functional and non-functional requirements from various stakeholders.

Domain scoping is one of the issues that should be taken into consideration when domain model is being created. Because process of generating assets that constitute domain model is expensive and time consuming, domain scope could be derived from economic aspects [25]. One of the ways of scoping a domain is to analyse needs of several stakeholders and “calculate their benefit function” [25]. Yijun Yu et al have suggested taking stakeholders’ goals into consideration when determining features that should be included into domain model [33]. If we do not have access to stakeholders, we still could use current applications from the same domain to identify their common features. It could be simply done by preparing a matrix where rows represent features and columns represent

different applications. By giving a point at each intersection of feature-row and application-column, the most common features could be identified. This method could be extended by using different weights for different features and/or applications. Similar approaches for identifying features for reusable methodologies have been presented in various publications [9, 31].

Determining economically and functionally optimal domain scope in some cases could be very difficult. When domain engineer tries to define in advance which features should be included in domain model and which should not, there is always a possibility that too many or too less features will be included. Those cases are called over- and underscoping [25]. When is overscoped the development cost increases as well as domain modelling time. From economical point of view in this case, company will not generate as much revenue as it could. In case of underscoping it could be difficult to create and deliver wide range of different products from particular domain because of lack of features. Such situation could force domain engineers to include new features into domain model during development cycle. Depending on whether missing assets were detected at early stage of product development or not, it could significantly increase development time of new product from SPL.

Economic aspects should be always taken into consideration when decisions about domain developing or domain scoping have to be made. Making decisions if domain should be developed or not, or how many and which features it should include is harder than making decisions about developing single application. Working on domain and creating environment for reusable software development is itself time and resource consuming. Benefit from implementing reusable practices can be seen in a long run term, thus decision for developing a domain or particular feature should be based on its usefulness in future projects and estimated revenue.

2.4 Software reuse

Software engineering requires from programmes, developers, system architects and other stakeholders involved in developing new software products, significantly large knowledge base when they are working on a new products. Because of that, non-trivial IT projects require both experienced engineers and significant amount of time and money to develop new application, add new features to existing one or adjust its properties to meet needs of different clients. Both academicians and industry experts are working on techniques and methodologies which would allow persisting existing “knowledge” in that field and shorten time of developing new products.

Software reuse is a paradigm which supports reusing once created design, components or implemented code in multiple projects. ”Software reuse involves generating new designs by combining high-level specifications and existing component artifacts” [26]. This is very challenging but beneficial activity in terms of costs, quality and shortening development time during future projects. All in all, the more products we could create based on reused components, the cheaper development process would be.

Discipline that is focused on providing solutions of mentioned concerns is called *software reuse* and is a very challenging and promising field for both academicians and industry experts. There are many software reuse concepts that are being constantly improved and adjusted to changing needs of industry. In general, purpose of introducing software reuse practices is concentrated on providing working products faster, at lower cost, with smaller number of developers and with higher quality. The main idea behind software reuse techniques is to using once created code or components, rather than writing them from scratch [21] – with respect to phrase “don’t repeat yourself”.

Software reuse approaches are aimed for persisting experience and knowledge base of domain experts and system developers in order to use it in multiple projects. Those activities are called *assets creation* and are done under domain engineering processes. Mentioned assets could be understood in multiple ways. Anything that is once created in order to serve as a base or an artifact of future projects from *same domain* or same family of related projects could be treated as such asset. Examples of those assets are:

- generic architectures,
- feature models,
- code generators,
- components,
- development practices and processes,
- frameworks and so on.

In software reuse approaches not everything has to be build in-house – it is also possible to buy *assets* for purpose of software reuse and integrate them into projects. Some approaches are actually very dependent on assets/components acquisition.

Implementing software reuse approaches does not come at ‘no cost’ and before engaging such practices for particular purpose, it should be evaluated if it is feasible and if it would yield expected benefits. Employing reuse oriented techniques or methodologies requires substantial amount of time, money and knowledge. The big overhead prior to product (or rather products) development is supposed to be compensated by facilitation of successive projects development. Because of that reuse approaches are suitable for those software products which are needed in multiple versions, variants or are marketed for multiple customers with individual needs.

There are various approaches and techniques for software reuse paradigm. They vary in terms of needed technical skills, integration level and expected outcomes. Different approaches also vary in organisational level, thus they require different roles and skills in order to operate. Brief descriptions and principles of selected approaches are given below.

2.4.1 Software product lines

A software product line (SPL) is a paradigm or form of software engineering practice focused on delivering software from reusable assets. SPL focuses on creating family of related products based on available components (also called assets) instead of creating everything from scratch. SPL is aimed to deliver diversity of products from the same family of products of from the same *domain*. Domain engineering principles, which are heavily used for SPLs have been discussed in previous sections.

There are various process models which could be used during usage of SPL. Following [26], those could be named:

- proactive
- reactive
- and extractive process models

Products created with proactive process model are based on pre-created core assets which are later on used to create final products. This requires significant number of developed assets and suits best those domains, which are well established and it is easy to predict in advance, which assets (functionalities) will be required. Another possible approach is creating assets “on demand, at the time when a particular functionality (asset) is needed. This is a reactive process model and suits cases when creating multiple assets in advance would be too

expensive or when it is difficult to predict all required assets for a particular domain” [26]. Last of the mentioned processes ”stays in between the proactive and reactive approaches” [26]. This approach is suitable when a company has already developed some working products that could be turned into assets and reused within SPL.

Assets accumulation and proper reuse is one of the key issues in SPLs. Depending on how many assets we have and how well could we integrate them into a working product could determine an efficiency of an organisation and could have an impact on operational costs and products quality. But there are other important issues such as assets granularity and their coherence level. These properties determine how well we can integrate and combine different assets (coherence) and how many different products would we assemble from set of assets (granularity).

Creating family of products requires specific approaches for analysing and modelling software products. Creating generic models and finding common features for applications from same domain enforces focusing on commonalities and variabilities instead of focusing on details. For purposes of analysis and discovering of key properties of software, Feature-Oriented Analysis approaches could be used. This would allow for discovering common and variable assets of products and would also allow for defining their relations and dependencies. For a purpose of representing variability and commonality within SPL, feature models could be used [8]. Feature models and feature oriented approaches are well established and widely used for representing variability, commonality, modules and different features of a software. Because of those properties, feature models could be successfully used for modelling product families in SPLs.

2.4.2 Component-Based Software Engineering

Component-based software engineering (CBSE) which is also known as component-based development (CBD) is another approach for software reuse in software engineering. Reusable assets in this approach are called *components* and are supposed to be independent from each other, provide well-defined services to other components or application and require very little or no customisation during integration process [25]. Components are also described by interfaces they are communicating with other components as well as with application. They suppose to be abstract enough to serve similar or same purpose within different applications.

Following [25], there is no single definition of a component and variety of authors proposed different definitions. For example, Szyperski et al. suggest that component is a piece of software that could be deployed independently from other components and are meant to be composed with different components through ‘contractual interfaces’ [28].

Because components heavily rely on interfaces and interoperability, it is essential to have components which are highly interoperable and have well defined and reasonable interfaces. Because some of the components are created for sale purposes, their creators must ensure that components’ interfaces will follow common industry standards. Examples of those standards are EJB, CORBA, RMI, HL7, etc. Such standards ensure that components will be able to communicate with other components or application without major problems. Except following communication and information exchange standards, there is also need to meet quality criteria for market targeted components.

When it comes to define quality of components, several aspects should be taken into consideration. Following [25], a *good component* should be:

- Well documented
- Cohesive
- Independent
- Useful
- Certified

Additionally, they should satisfy the several properties [25]:

- Should be composable
- Should have well-defined interfaces
- Conforming to a component model
- Should be secure

In practice, CBSE could be difficult to fully implement because of some technical issues, especially if components from various third party vendors are used. First and most obvious issue for component integration could be interface compatibility with other components or whole application. Even when the documentation is detailed, there is always a risk that some peculiarities of component have not been documented and integration could be bothersome.

In recent years an emergence of Web Services (WS) and products following Service Oriented Architectures (SOA) becomes more noticeable. Such services could be compared to components in terms of exposed interfaces and being independent from each other. The biggest difference between service and component is their allocation. When component is used, it have to be build into application it serves while web service could be executed on provider's server and just provide its service instead of being physically delivered to customer or integrator.

2.5 Feature modelling

Increasing complexity of software products in terms of used components, libraries, and written lines of code requires from domain engineers and system architects new approaches toward software engineering, structuring and documenting. Software design could be presented at various levels of details and each level could serve different purpose and be addressed to different stakeholders. When it comes to analyse, design or maintain complex programs, analysing them at the level of class diagrams could be inefficient, time consuming and error prone. In such cases, for some purposes it would be worth to consider introducing *features* as an approach of analysing and documenting software products.

There are a considerable number of papers treating about various usages of features and about different views on those features by authors [8, 13, 12, 25, 31]. Because of that, different definitions of features could be found. Orla Greevy et al. suggest that “a feature represents a unit of domain knowledge, and it typically corresponds to a realized functional requirement of a system” [13]. Other authors write that “feature is an abstract description of a functionality given in the specification” [32]. It could be concluded that a single feature could be spread across many classes and one class could serve more than one feature. As [4] suggests, it creates n-to-n relation between them. Basing on that, we can conclude that features provide higher level of abstraction than classes, thus modelling or representing system in terms of *features set* or *feature tree* would be more understandable for variety of stakeholders. This property in system representation is very important when someone have to face very complex system, composed of hundreds or thousands of classes. [32]

Features could be introduced at various stages of software products development, thus will play different role and have different purpose. Some authors suggest introducing features at the domain analysis stage [14, 19, 23], which would facilitate product development and analysis. But that is not the only possible use of features – other authors advocate using features for tracing changes within software projects and analyzing its evolution [13]. Another possible usage of features is facilitating maintenance of legacy software. As Norman Wilde et al. have noticed that there are various methods for locating features in legacy software [31]. When it comes to introducing improvements of bug fixes, such approach makes job of software engineers easier, especially in the case of complex software products. Having features and their mapping to classes, would allow for fast identification of relevant pieces of code when a particular feature should be updated or modified.

2.5.1 Feature trees and feature models

Features could be used for representing whole models of software – by feature models. Basic idea behind feature model is to represent variability and commonality of a software product in a compact way. To do so, features are organised (most of the time) as hierarchical- tree-like structures, called feature trees. Feature tree could be used to represent software model or even whole domain of a product family. This approach is especially useful for representing models or domains for reusable approaches. There is no consensus about one common notation for representing feature trees and different researchers proposed their own notations. P. Heymans et al. [16] have described several ways of representing feature diagrams and described properties of different notations.

In recent years, much attention has been paid to potentially sensitive areas of feature modelling [23], those are:

- How to identify features?
- How to represent feature in a structure?
- How to identify constrains, dependencies and relations among features?
- How to assure that feature model is complete, consistent and valid?

If we think about feature tree which contains more than thousand features and such feature model was evolving for more than few years, it becomes easy to image how difficult it could be to manage such design without appropriate methodologies and approaches. The problem could emerge when we have to introduce a new feature to the feature tree or when we have to modify existing one. Doing so, we have to be sure that new feature (or removed one) will not break consistency of a feature hierarchy. One way of specifying static dependencies between features is introduction of constraint relationships. Unfortunately, “constraints only describe static dependencies between features, but tell little about how these features interact dynamically with each other at run-time” [23]. Hong Mei et al have proposed a metamodel for addressing issues of both static and dynamic dependencies between features [23]. Their metamodel:

“(...) defines three important relationships between features, namely refinement, constraint and interaction (...) The refinement provides a way to explore various system features from high levels to low-levels of abstraction, and to organize features as hierarchy structures. The constraint provides a way to specify static dependencies between features. And interaction provides a way to express dynamic dependencies between features”

Obviously, even having such metamodel for representing feature-based domain models, automatic approaches for consistency and validity checking of derived products are required when it comes to work with very large domain models.

2.5.2 Feature oriented domain analysis (FODA)

Domain analysis is a crucial stage of developing software thus quality of products and possibility of achieving venture's goals are strongly related on process of domain analysis. The purpose of this process is to both, better understand domain space as well as represent it in a way that other stakeholders could understand what the purpose of a domain is and how the domain is constructed. Domain analysis is a process within domain engineering activity and focuses on analysing domain and its properties for a particular purpose. When it comes to reusable software a proper approach for domain analysis should be taken into consideration. Analysing a domain for purpose of reusability requires support for variety and commonality – in this way, assets for SPLs or other approaches could be created.

Feature Oriented Domain Analysis (FODA) is an approach which meets mentioned requirements. An analysis process leads to discovery and identification of features representing domain where “Features present customer valuable properties of systems” [25]. Feasibility study of FODA was released in 1990 and since that time, that method has been actively used by many researchers [10, 20, 33] working on different aspects of software engineering. It was proposed as “a method for discovering and representing commonalities among related software systems” [19]. For a purpose of software reuse, it is very essential to find common and variable aspects of a domain within a family of similar or related software products. If we could achieve that, it would be easier to identify features which should be included into SPL as mandatory or optional ones.

Treating a domain in terms of tree-organized feature models allows to model hierarchy and map connections between different features. Representing features as a tree makes the representation of a domain clear and easy to comprehend.

Additionally, such representation could be considered as an asset and reused for different projects within the same domain.

Finding features during analyse of multiple products is not an easy task. It could be asked how to determine which features should be optional, which should be mandatory and which should be omitted. It is not an easy question and in most non-trivial cases, extensive domain expert's and software architect's knowledge is required. Ilian Pashov et al have proposed a statistical approach to find out, which features should be and which should not be included into feature tree [25].

FODA could be used as for domain modelling purpose where domain model could be represented as a feature tree. "Feature modelling is a method originally developed for structuring domain properties from customer's point of view" [25].

2.5.3 Feature-Oriented Software Development (FOSD)

Feature-Oriented Software Development (FOSD) is a paradigm which has emerged from different disciplines in software construction and reuse. FOSD is aimed to serve different purposes such as construction of software, its customisation and composition of systems. In FOSD everything is focused on a concept of feature and feature models which are a high level abstraction of applications and serve as description of problem space that software system should address. By focusing on a feature, system could be decomposed and structured in a way, which allows for separating most essential parts of application and possibly restructuring such applications to serve different purposes. As Sven Apel et al points out:

“The goal of the decomposition is to construct well-structured software that can be tailored to the needs of the user and the application scenario. Typically, from a set of features, many different software systems can be generated that share common features and differ in other features. The set of software systems generated from a set of features is also called a software product line.” [3]

In FOSD it is favoured to use concept of feature during all phases of software development cycle such as analysis, design and implementation. [3] By focusing on features during all phases, the design and implementation are supposed to be exactly mapped and meet the requirement. Such exact mapping would allow assuring that during transition between different phases the original requirements will not be misunderstood or wrongly interpreted. Additionally, focusing on features not helps to focus on consistency but also facilitates identification of commonalities and variabilities which makes implementation of software reuse techniques easier.

Feature-oriented programming (FOP) is another approach related to FOSD which focuses on usage of features in software development. FOP approach focuses on modularising features instead of allowing the code to be distributed across different classes [27]. Sven Apel et al are also trying to use feature-based approaches for software development to develop systems based on service-oriented architectures. In their work [2] they point out that “formal foundation of feature orientation provides a straightforward way to set up a formal specification and type system for services based on features, not just on interfaces.” Works on FOP and designing service-oriented applications basing on features shows high potential of incorporating features in variety of areas of software engineering.

CHAPTER 3

USING FEATURE MODELS FOR REUSABILITY IN AGILE METHODS

In this chapter, an approach for using feature models for reusability in Agile Methods and code retrieval tool is introduced. Until this point, related works and technologies which were presented would allow for better understanding of proposed methodology. This chapter starts with brief overview of proposed methodology and part by part will explain fundamentals and basis of it. After introduction part, constraints and limitations will be mentioned with explanations how such limitations could be overcome. Later on, suggested usage of the approach will demonstrate its real strengths and will let readers to judge usefulness of it. In order to benefit from the approach, a dedicated tool was created for purpose of evaluation – tool for source code retrieval basing on predefined features. Combination of the approach and dedicated tool for source code retrieval makes a fully functional system that could be used for source code reuse and re-engineering purposes. The tool which has been developed will be presented and described in last part of this chapter. Basing on that description, similar solutions could be designed in order to meet specific needs of particular development environments.

3.1 Overview of the approach

Presented approach is combination of feature based software design and agile development methods. It has been inspired by agile development techniques which are lightweight, developer friendly and which are well balancing effort needed to use it and a quality of final product developed under guidelines if it. Such way of thinking about software reuse is in contradiction to some of well known software reuse methods such as software product lines, code generators or component based development – which are quite heavy and difficult to introduce in terms of initial costs and effort. All of those methods are complex to implement and require high overhead in terms of creating or acquiring assets (code generators, components, etc). However, on the other hand, mentioned “heavy” approaches towards software reuse in particular situations could allow for *faster* implementation of particular function or feature in some cases when already existing components could be easily integrated into working software. It should be mentioned, that even *if* one of those approaches would allow for faster implementation of part (or whole) of the application, most probably additional effort for integrating or testing “reused” component should be done anyway.

One of the very noticeable aspects of the approach is that it allows to access low level resources via highly abstract description of the project. Source code of the application is indeed very low level resource and representing it with feature model, which is in fact more abstract than UML class model makes the description very general. Making description (or document) layer of the approach as an abstract one, developer will require less time to access resources he/she needs. Of course, reuse process is not automatic and requires involvement of a developer, but by accessing reuse assets consciously, developer can tailor and modify the reused code to specific needs of a new project.

Proposed approach for code reuse is based on two fundamental aspects. First, a feature model representing source code (or application) is required. Second, for each feature (of a feature model), at least one “top level” function should have been identified in advance and marked with special Java annotation. *Top level* function is understood as a function which is both, most important and probably called as one of the first during feature execution. Under some circumstances more than one such function could be associated with particular feature and each of them should be adequately annotated. Examples of how to identify such top level function and how to properly annotate them will be presented in “Creating sample project”. Proposed approach requires access to whole source code of analysed and processed application.

As mentioned earlier, this approach imposes usage of feature models as a representation of particular application. Such imposition is justified by the assumption, that high level code representation would allow for more efficient source code reuse practices. In this way, when developer or other team member wants to access source code which represents particular feature of feature model, such person having appropriate system (presented in this chapter) can do it in few clicks of a mouse. When an application is represented by a feature model, and another similar application is created, developer can find source code of features easily and it gives the development team the advantage and possibility of not rewriting the code. Such approach could be very efficient and is fairly easy to implement and still gives full control over the reused code to programmers.

Proposed approach was created to support agile development methods because it was designed to be lightweight and easy to use. The most obvious and demanding aspect of using proposed approach is to create and maintain feature model for each project and annotate appropriate functions within code. Such

restriction could be perceived as beneficial because application could be rapidly modelled and feature model could be treated as additional documentation of the project and because of high level of abstraction, such documentation could be useful for various stakeholders. Also, by creating feature model, developers have to think of code they are writing in terms of features which could lead to writing more coherent code— as developer tries to write code within bounds of particular feature.

What in practice does it all mean? In practice, by connecting source code with a feature developer is able to retrieve whole (or almost whole) source code responsible of execution and functions of particular feature. Easy access to such code could lead to fast implementation (or instant implementation in some cases) of same or similar features within separate projects. Of course, someone could say, that developer can retrieve source code without any tool by simply analysing and reading it. That is true but if developer will face need of analysis one or more projects, each of them having more than 50000 lines of code then such analysis will be very time (resources) consuming and of low benefit in terms of reuse. However, if each of those projects would have well created feature model, and each feature would be connected with source code then code analysis and code retrieval would take much less time. By taking less time, it becomes obvious that such approach, if properly used, could shorten time of development cycle and decrease development costs.

Besides benefits mentioned above, there are some other, not direct and not obvious ones. At the time when developer is looking for a source code responsible for a feature, he/she is at the same time doing code review. This could lead to two potential benefits. First, there is a chance that during such review some bugs will be spotted and fixed which is obviously beneficial. Second, during review a developer, especially inexperienced one can learn

something new, especially if the code which he/she is reviewing was written by experienced programmer.

The figure 3.1 is a simplified overview of how the approach is used when a new project is created. However the diagram is simplified, it could be noticed that the approach is pretty straightforward and not complicated. All what is need to be done is to create a feature model reflecting a real project and annotate appropriate functions representing *entry points* for each of the feature. Of course, usage of the approach is not constrained only to *new* project and feature model and links between features and code could be applied to existing ones as well.

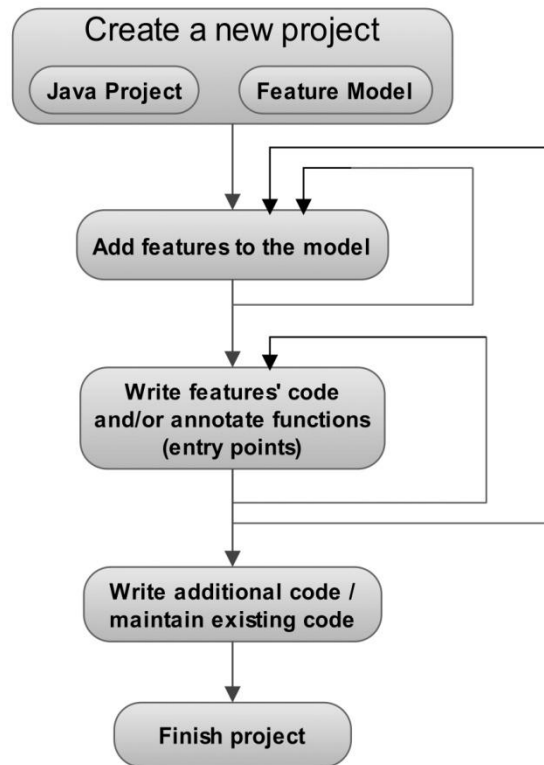


Figure 3.1: Overview of creating project with proposed methodology

Creating a project with respect to given approach is only one aspect of it. Another, probably more interesting is how the code retrieval process works. What is obvious, in order to retrieve a source code of a feature, first we have to have at least one project annotated and enhanced with feature model according to guidelines of proposed approach.

First step in code retrieval is to identify a project or group of projects which potentially could have implemented a feature we are looking for. After that developer is able to view each feature model and try to identify feature he/she is looking for. Source code of particular feature could be easily accessed by clicking on a feature via graphical interface. After that, developer can navigate

via tree of functions which participate in execution of feature – in this way, developer gains access to source code of functions which participate in feature execution. This process has been presented on Figure 3.2.

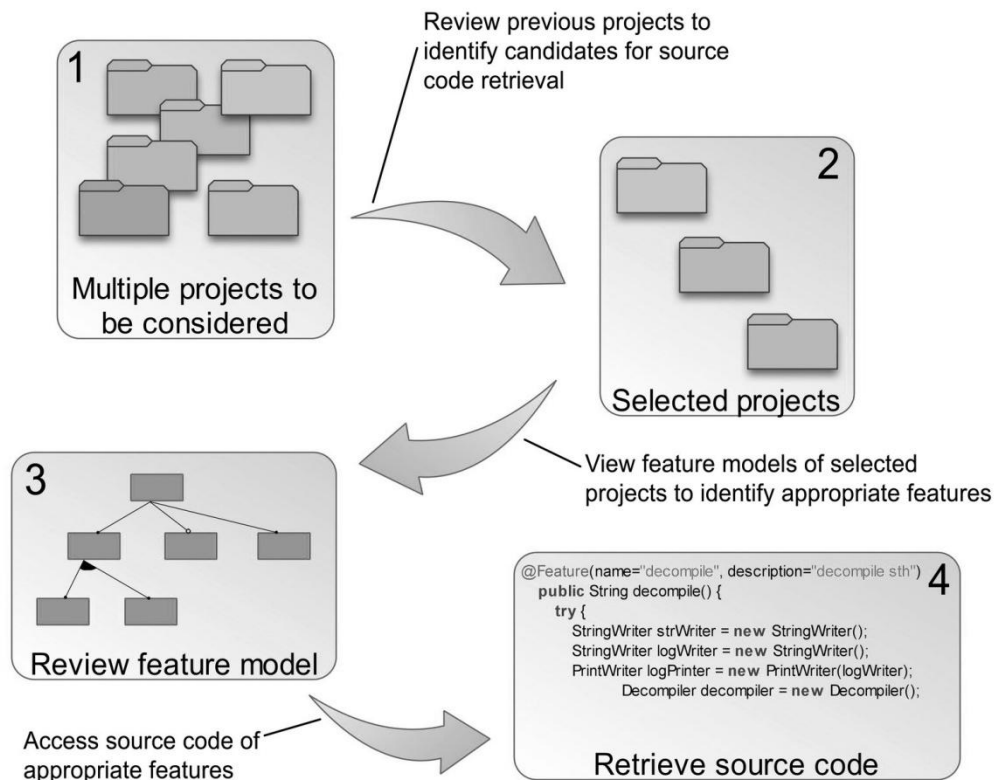


Figure 3.2: Process of identifying assets and source code reuse

What is also interesting is how exactly the retrieved code looks like? Let's assume we are writing software for controlling an engine and we are interested in code responsible for starting it. In modern cars the process of starting an engine is quite complex and involves heavily usage of on-board computers which are controlling this process. For sake of simplicity in the given example we will consider bit simplified process, where just one cylinder is taken into consideration:

- Pre-start checks:
 - Check security code
 - Check engine temperature
 - Check oil level
- Start fuel pump
- Calculate air-fuel proportions
- Inject air-fuel mixture
- Compress mixture
 - Rotate crankshaft to appropriate position
- Ignite spark plug

Now, let's assume that simplified source code of above process looks as given below:

```

package com.carsofttronix.engine;
public class OperateEngine {
//init devlarations...
  @Feature(name="Start Engine", description="...")
  public void startEngine(Code secCode){
    if(!generalCheck.preStartChecks(secCode)){
      return;//pre start checks failed
    }
    fuelPump.startFuelPump();

    int i=0;
    while(!isEngineOperating() && i++<500){
      FuelAirMixture fam =
engineComputations.calculateStartFuelAirMixture();
      injectFuelAirMixture(fam);
      compressFuelAirMixture();
      igniteSparkPlug();
    }
  }

//some other functions ...

```



```
}
```

Let's also consider below code as a part of example:

```
package com.carsofttronix.engine.checks;

import com.carsofttronix.model.Code;

public class GeneralCheck {

    public boolean preStartChecks(Code secCode) {
        if (!checkSecCode(secCode))
            return false;
        if (!checkEgnineTemperature())
            return false;
        if (!checkOilLevel())
            return false;

        return true;
    }

    private boolean checkOilLevel() {
        // do some calculations and return true or false
        return false;
    }

    private boolean checkEgnineTemperature() {
        // do some calculations and return true or false
        return false;
    }

    private boolean checkSecCode(Code secCode) {
        //do some calculations and return true or false
        return false;
    }
}
```

As we can see, the *top level* function `startEngine(Code secCode)` is annotated with annotation `@Feature(name="Start Engine", description="Feature responsible for starting engine")`. The

function does not itself do everything required to start the engine but calls other functions responsible for appropriate tasks – as it supposed to be in Object Oriented programming. Now, when we will require retrieving the code responsible for starting the engine, whole source code will be returned – not just *top level* function. No matter how nested the function calls are, the tool for retrieving source code basing on features will find them and printout on a screen. In this way, developer will have “one-click” access to source code of all functions which are involved in starting of the engine. In the given example, such functions as `preStartsChecks (Code secCode) , checkOilLevel () , checkEngineTemperature ()` or `checkSecCode (Code secCode)` will also be returned, and as we can see, they do not have to be directly annotated with `@Feature`.

Analysing and processing source code and somehow determining runtime behaviour (identifying functions which will be executed during runtime) could be challenging. Java as an advanced Object-oriented programming language provides for programmers wide range of programming mechanisms such as polymorphism, inheritance, overloading, abstract classes, interfaces and anonymous classes and interfaces. Those advanced mechanisms make the language more efficient for the programmer, but it becomes more difficult to analyse the code, as runtime behaviour cannot be always predicted. As Dr. David Binkley wrote in his article [6]:

“Against an increasing need for higher precision source-code analysis, modern languages increasingly require tools to handle only partially known behaviour (in the case of Java this is caused by features such as generics, user defined types, plug-in components, reflection, and dynamic class loading). These features increase flexibility at runtime, but compromise static analysis.”

3.2 Constrains and limitations of current work

For a purpose of current work, a tool that makes usage of proposed approach possible has been implemented. This tool is limited to work with Java source code, version 1.5 or higher. The requirement for Java version belongs to the fact that annotations which are required to identify features within code, did not exist prior to Java 1.5.

In theory, the approach could be used with any Object Oriented language. For that purpose, a parser and code analysis tool should be implemented. Basing on this work and explanations how proposed tool was designed and implemented, implementing similar tool for other object oriented languages is possible.

Some of the limitations of current tool for code retrieval belong to the ways Java programs could be constructed. For example, when we want to identify source code which is responsible for functioning of particular feature, we have to recursively follow execution of each nested function call within a *top level* function and within each of that nested function. In order to do that, we have to know which specific class the function is called from. That could be complicated if at one level we will not have information about concrete class but instead we will have information about *interface* of such class. There could be a case when object of concrete class is assigned to reference of an interface during runtime of the program. In such case, it could be not possible or it would be very difficult to identify during source code analysis 100% of source code that constitutes features. Such constraint could be possibly bypassed with satisfactory results in most of such cases with usage of advanced analysis techniques or some heuristics which would give *possible solutions*. Such solution goes beyond current work and is not necessary to affirm if proposed

approach meets its purpose – if it allows reusing source code easily and identify source code for modelled features.

Despite mentioned limitations, proposed approach could be evaluated and it could be judged if it meets expectations in terms feasibility of code reuse and retrieval. After such judgements, improved tools could be implemented to meet higher standards and needs of enterprise-level development environments.

3.3 Suggested usage of the approach

Introduction of any approach into a working environment does not come at no-cost because it requires time for adaptation, possibly training of developers and some kind of overhead during its usage. Before an approach is introduced into working environment and adapted by developers it should be evaluated if it will bring expected benefits in terms of increased quality of software products, reduced development cycle and/or reduced cost of development.

There are no silver bullet solutions or ‘one approach fits all’ so in this part of the thesis, suggested usage of the approach will be presented. This should bring potential adapters closer to a decision, if presented approach would fit their needs and match their environments and working style. Also, by presenting how approach could be used, reader could get some ideas of possible benefits and good practices implied by the approach.

Usage of proposed approach could be divided into two distinctive activities. First of them could be described as *assets creation*. That activity constitutes creating feature model for each project we want to reuse in a feature and linking features form the model with source code. That activity could be executed during development of each new application *or* could be also carried out on old projects which company or group of developers have been working on

previously. Another activity is retrieving and reusing once written source code. In order to perform following activity, former one should be first executed. Also, the more projects have been done or adapted to proposed approach, the more code to reuse developers will have. We could assume that costs of developing applications will systematically decrease with number of project performed under guidelines of source code reuse approach.

3.3.1 Roles and responsibilities

In order to adapt an approach, some additional effort should be made. If benefits of adapting the approach are higher than cost (effort) then adapting a approach would make a sense. Proposed approach requires introduction of additional roles and responsibilities into development team. Those roles and responsibilities are not very absorbing so successfully could be assigned to members of development team – without need of additional personnel.

First of the roles is *feature model modeller and maintainer*. Let's call such person *feature model owner*. Introduction of feature model is a must, and there should be a defined person who will be responsible for it. The responsibility includes developing a feature model, assuring it is consistent with project requirements and source code. Maintenance involves keeping the model up to date and when required, assuring that source code is well linked to such model. If needed, such person should point out when source code becomes not consistent with the model and should have enough authority to demand from developers to improve or refactor their code to match feature model. Such role should be assigned to an experienced developer who is familiar with feature modelling and architectural issues. At the end of each project, *feature model owner*, should additionally check if code retrieval of features gives reasonable results and if features are correctly linked to the source code.

Each member of development team, in context of proposed approach, could be named *feature model user*. Responsibilities of feature model users include writing code which is coherent with feature model proposed by feature model owner and gives them permission to reuse once written code during previous projects. Also, feature model users are obligated to report, and if possible, fix (or report) bugs discovered during code retrieval operations. It is recommended, that when a developer reuses code from one project, he/she makes an appropriate comment in a new project, indicating project and feature from which the code has been taken. Such activity would allow other developers to find commonalities between projects which should lead to higher awareness of different projects available in the reuse repository.

It should be also mentioned, that because source code and projects are meant to be reused and analysed by various developers in different period of time - sometimes, months after completion of a project. This fact should motivate developers to write code of higher quality, especially if they will be aware of possibility of linking them with code they wrote, so their reputation can be affected by low quality code. Such linking and identification is not a part of proposed approach, but could be easily achieved by putting “author” in JavaDoc comments. Of course, if developers will place their names in JavaDoc comments or not, is another issue and it should be ensured by development culture or policies.

3.3.2 Code and feature models sharing and maintenance

Sharing source code and feature models between developers and even among different teams in case of large organisations is essential in this approach. As it was mentioned, one person is responsible for particular feature model of a project, so just one person should be able to modify and update it. However, everyone who is participating in process of developing software in particular

organisation should have full (read) access to the code and feature models. Sharing code between developers is not an issue since there are various revision control systems which are meeting their purposes perfectly. However, in case of sharing feature models other approach should be considered. First, only *feature model owner* is allowed to modify feature model thus sharing feature models over such systems would be redundant. Another issue is that to each feature model, a source code of an application should be available. Because for code reuse purpose most of the time we will be considering finished projects, where no regular code updates are made, then such code also does not have to be shared via code repositories because there should be no modification and there would be no need for version control.

Taking into account above consideration, it could be concluded that most appropriate approach for sharing feature models and code would be to give to all interested participants instant access to most recent version, without need of synchronisation. As also mentioned, different groups of participants should have different privileges – feature model owners should have full read-write access to their projects while feature model users just read access. Such environment could be established basing on solutions such as directory service (i.e. Active Directory) or other forms of sharing files among different user via LAN. Of course, such solutions are not restricted only to LAN networks – resources could be remotely accessed via VPN connections.

There could be a situation, when a significant change has been made in project's source code so in such situation, to keep the code in the directory service up to date, feature model owner is responsible to update it when significant changes are made. Such approach will make code available for reuse possibly consistent and still it will be possible to update it. Also, project does not have to be finished in order to be eligible for reuse. Feature model owner who is

responsible for both feature model and its relation to the source code at one stage could decide that enough features are implemented and project under development could be shared for reuse purpose. Following figure demonstrates various activities and processes which could occur during usage of proposed approach.

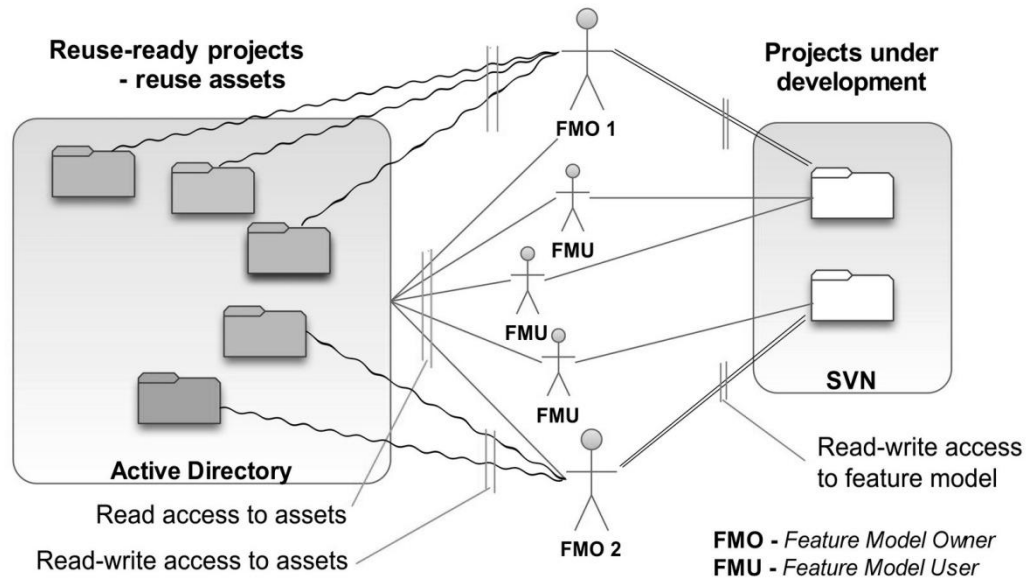


Figure 0.3: Roles, assets and project allocation and access for proposed

3.3.3 Feasibility of the approach

Basing on description and purpose of approach, its feasibility for particular purposes could be at least hypothetically judged. Knowledge of how an approach will be used and how development teams are working in production environments is essential for judging feasibility of an approach in different cases. Of course, it is hard to predict how an approach would be used in every kind of environment and by different development teams with exact precision. However, basing on several general examples it should be possible to judge if an approach is feasible for various cases.

For purpose of demonstrating feasibility of proposed approach, several cases when the approach would meet its purpose and when it would not meet will be presented. Each case gives also insight on purpose of developing the approach. By analysing them, reader can with a certain level of confidence conclude if the approach would work in his/her environment.

Case 1:

In first case we are considering usage of proposed approach in a small or medium size company writing custom applications for its external customers. There is one or more projects developed at the same time and projects are manifesting at least minimal level of similarity – for example, project are from similar domains – banking, accounting, Business Intelligence or at least have similar form – web applications, GUI applications, embedded applications and so on. Projects could rely on external libraries, but still, significant amount of programming is done by developers.

In the above scenario introduction of the proposed approach should bring expected results for benefiting from reusable assets. In such environment, we are able to create assets – develop feature models and link features to the source code. In this way in further project developers could use source code from previous (or current) project by easily retrieving and reusing it. It is important to notice, that there is at least minimal level of similarity between projects – in such case, there is chance of need for reusing previously written code. What is also important is number of projects developer annually – the more are developed, the faster assets would be created. It means that with larger number of projects developed ‘in house’, benefits from adopting the approach will come sooner.

Case 2:

In next case, a large and distributed organisation working on variety of projects is considered. It is possible, that the organisation has offices in distinct locations and projects are varying from small to very large. In terms of domain and type, applications are also diverse. Annually, many projects are completed and hundreds thousands lines of code are written. Applications are developed for external customers, internal use or mainstream market.

This is bit more complicated case in compare to the first one. In this case, we are considering very large organisation which develops diverse range of products – in terms of customers, domain and type. Such high diversification is not a perfect case for code reuse. However, if we take into consideration that annually a lot of different projects are completed, then there is a high possibility that sufficient level of similarity between some of the projects will occur and source code reuse will be beneficial. At this point, it should be also mentioned that the larger organisation is, the more difficult and costly introduction of the approach would be. For example, in order to provide access to projects and source code, VPN network should be established – as it was mentioned earlier probably sharing feature models with theirs source code via code revision systems will not be best solution. Another point is that having a very large number of projects as source reuse assets, it would be reasonable to somehow organise them into ontology to allow developers for faster access to feature models they would be interested in. Such ontology goes beyond this work and is suggested as a further work. Despite mentioned difficulties, possible benefits from ‘economy of scale’ should make this approach feasible for large organisations.

Case 3:

In this case, a small or medium size company is considered. Our hypothetical company develops software for much specialised markets such as military or

banking industry. It could be considered a good case for source code reuse – company is focused on very narrow market, so we could guess that there will be significant level of similarity between projects. However, because of confidentiality or licensing issues, code cannot be easily shared between not involved developers.

In this case, there are issues of licensing or confidentiality which obstructs possibility of sharing and reusing source code. In such case, obviously the approach has to be used with caution. In given example, other software reuse techniques could be more appropriate: for example using code generators.

Case 4:

In the last case let's consider a small, medium or big company which main activity is to integrate third part's components and libraries. We can assume that composed software is from same or few very similar domains and there is some level of similarity between different products. Such integration involves some level of programming and for each final product some amount of source code has to be written.

In this case, company already has adopted a technique for software reuse – in this case software products are created from ready, market-available components. However, it was also mentioned that developers are writing their own code to integrate components and libraries. In such case, there are few aspects to consider. First, how much of source code can we reuse from different projects? Second, how much benefit could we have from having additional documentation and description in form of feature models? Answering first question, how much source code can we reuse would depend on how much source code have been written for each project. Even if little code has been written, such code could have valuable bits of information about integration

process which could allow developers to integrate some components easier while working with them next time. When we think about benefits from having additional documentation and description of a product it becomes obvious that such feature model would allow different stakeholders to better understand the project. Even when project will be finished and archived, coming back to it and understanding how it was constructed would be easier when feature model would be available. Of course, there could be some problems with linking features to source code. For example, if functionality of particular feature will be fully delivered by component or library, then we simply cannot do that. We could just link feature to the source code our developers would wrote. But still, impossibility of linking feature with source code does not mean we should not create a feature model. In this example, it should be judged individually if potential benefits would be worth to introduce such approach into working environment.

Above cases could give some insight into feasibility of the approach in different environments. There are also other factors which were not considered and could have potentially big impact on successfulness of approach introduction. In this case, experience of developers could be significant. Developers should have enough experience to be able to work with source code which someone else wrote which requires analytical thinking. Also, developers should be able to think about code they are writing in terms of features and deliver coherent and feature relevant code.

In advance, it could be difficult to be sure if proposed approach will work out in particular situation or development environment. However, basing on presented cases at least up to some level of certainty we should be able assess usefulness of the approach. Taking it into account, if one decides to introduce any approach into production environment, such introduction should be gradual in order to

evaluate how it meets demands and expectations. Also, by gradual introduction of an approach, it is more likely that it will be accepted by developers because gradual introduction of it, will not meet equally high resistance as in case of full introduction to every team and every department.

3.4 Design of the system

In order to efficiently use proposed approach a dedicated tool for retrieval of source code linked to feature model has been developed. This part of the chapter explains how the tool works and presented information could be useful in case of extending proposed tool or writing a new tool for different programming language. The approach with support of that tool constitutes a system for source code retrieval and reuse basing on feature models. The tool could be divided into several components:

- Code analysis
- Feature model representation
- Code retrieval
- Code visualisation

A part of code from another work on Feature modelling has been incorporated into the tool [29]. The code which has been included from that work allows for displaying and drawing feature model. Also, a Java parser which generates AST of Java code, has been used in this work [1].

Each of those components could constitute separate tool and in fact, many open sources applications have been used during development of presented tool. Because each component of the tool highly independent, extending or changing some of them would be possible. As it will be proposed in ‘Conclusion and

further work' there are several possible ways of improving and/or extending delivered tool.

One of the main aims during designing and developing was to deliver an application which will allow evaluating and testing proposed approach. Besides that, the tool was also designed to be user friendly, simple and easy to use. The tool has been written Java programming language, version 1.5.

3.4.1 Code analysis

Code analysis is one of the very first processes during runtime of proposed tool. Code analysis is executed once the project is created/loaded into the application or on demand – for example user can execute action allowing for analysing code and refreshing features representations. It could be done when user knows that source code has changed and it would be reasonable to work on fresh data.

Code analysis is divided into several steps and requires usage of several structures (classes) implemented as a part of the tool:

- Storing in memory representation of Java files (“SourceRepresentation”):
During this process, all Java pointed by the user during project creations are being discovered; their locations and file content is then stored in special class (structure) for easing access during further stages of code analysis.
- Preparing and storing in memory “CompilationUnitInfo” of discovered Java files:
CompilationUnitInfo is a class which directly represents discovered interfaces or classes. It has to be separated from “SourceRepresentation” described above, because single source file can represent multiple classes. During initialisation of CompilationUnitInfo, such information

as package, class/interface name, imports declarations, field declarations and some more detailed information about interface/class are identified during analysis and stored in memory. After that, objects of this class are grouped by packages in which they are declared. During further stages of analysis, it allows for easier access to classes/interfaces of the same package.

- Discovering and storing in memory “FeatureRepresentations”:

This stage of code analysis requires a lot of parsing of given source code. For that purpose an external Java parsing library is used. First phase of features discovery requires identification of functions annotated with “@Feature” annotation – so called *entry points* which are in fact representing those functions which are executed as first during execution of particular feature. Those functions – entry points will be used during further analysis and discovering of functions involved in functioning of given feature. One of the outcomes of this stage of analysis is collection of *BasicFeatureInfo*. This class constitutes feature name, method name, method body (method declaration) and information required to identify class (from analysed source code) in which particular feature was discovered. Basing on those *BasicFeatureInfo[s]* entry points will be later on identified and stored in memory.

- *BasicFeatureInfo* clustering and entry points initialisation:

This is the last stage of initial identification of features. After identifying all *BasicFeatureInfo[s]* they are clustered into appropriate representations of features (class *FeatureRepresentation*). It is required, because a single feature could be annotated at several different functions thus *FeatureRepresentation* could be composed of several *BasicFeatureInfo[s]*. After all *BasicFeatureInfo[s]* are allocated at appropriate *FeatureRepresentation*, collection of *FunctionRepresentation*, which is in

fact a collection of entry points, is generated. At this stage, FeatureRepresentation is represented by feature name, feature description, set of BasicFeatureInfo and set of FunctionRepresentation (entry points).

Those are initial stages in code analysis and all information is stored in memory during runtime of the system. Now, we know which features are present in a source code we are analysing and processing but the source code of each feature has not been identified yet. The source code identification for each feature is done upon user request – in provided system via graphical interface. How does it work and how source code is retrieved will be described in further part of this work.

3.4.2 Representing Features and Functions

Feature representation is essential for proposed tool. It is required for both, providing right data structure for making it possible to visualise feature model as well as make it possible to represent information required for identifying and providing source code which represents the feature. It is also essential to understand how such representation is created during runtime of the system. Generating representation is related to code analysis which is described with more details in further part of this work.

In proposed solution features are represented in two separate ways. The reasons for such form of dualism of features representation belongs to the fact that tool for drawing feature models is based on a third part source code and it has its own representation for feature models. For purpose of source code retrieval of a particular feature, independent model for representing features has been proposed. Someone could argue that it could be better to have one unified model for representing feature model, but despite the fact that those two models are

representing features, those models are having completely different purpose in the system. One of them makes visualisation of feature models possible, while another makes it possible to retrieve source code thus having unified model would make it unnecessary complex.

For better understanding how the system was designing, explaining how feature representation for code retrieval was designed is more crucial. By understanding how feature representation is modelled and which roles particular parts of the model play will allow to better understand the system and process of code analysis.

3.4.3 Code retrieval

Code retrieval is most crucial and complex module of provided tool. As it was explained in “Overview of the approach”, retrieved code does not just contain code of annotated function but also source code of all functions which are called as a direct result of calling annotated function – direct call for example does not include aspect oriented function executions.

How does code retrieval work in practice? First step is to properly process source code to get proper representation of *FeatureRepresentation* and *FunctionRepresentation* as it was described in previous section. In order to prepare it, an inner representation of source code in parser format is prepared. The parser can traverse such representation and identify every single element of source code (declarations, variables, conditions, function calls, etc). With such parser, the source code is searched for function declarations and for each method, it is checked weather the function is annotated with annotations or not. After identifying such function, it is checked weather one of the annotations is the one we are looking for - @Feature. Such lookup is done for each source file pointed out as part of the project. It should be mentioned that particular feature, for

example `@Feature(name="startFuelPump", description="...")` can be annotated to more than one function thus during discovering, existing internal representation of feature is extended with newly discovered functions. Because multiple functions can be annotated as the same feature, thus before source code for a feature is analysed for code identification and retrieval purpose, it have to be assured that all annotated features are discovered prior to such analysis. After the process of analysing all source files is completed, the real source code identification and retrieval for features can begin.

Next stage in source code discovery for feature is identification of function call hierarchy where function is represented by mentioned earlier `FunctionRepresentation`. Figure 3.4 represents simplified model of `FunctionRepresentation` with parent-children relationships hierarchy. It is important to know that in order to better understand how code retrieval works.

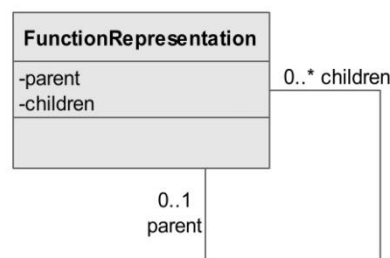


Figure 0.4: Simplified model of function representation

During analysis of function’s body, parser navigates through the abstract syntax tree (AST) looking for function calls declarations in order to prepare such parent-children hierarchy. Before details of analysis are revealed, let’s consider following functions (Figure 3.5, 3.6, 3.7) samples as possible input for process of source code analysis.


```

@Feature(name="first feature", description="...")
public void easyFunction(){
    this.functionCall();
    super.anotherFunctionCall();
    doSomething();
}

```

Figure 0.5: Sample Java function - subject to source code retrieval and analysis, 1

```

@Feature(name="second feature", description="...")
public void function(){
    FooClass fooClass = new FooClass();

    fooClass.callFunction();
    fooClass.callFunction(new Integer(1));
    int i=1;
    if(fooClass.checkInteger(i)){ }
}

```

Figure 0.6: Sample Java function - subject to source code retrieval and analysis, 2

```

@Feature(name="third feature", description="...")
public void complexFunction(){
    FooClass fooClass = new FooClass();
    BooClass booClass = new BooClass();
    IntegerGenerator intGenerator = new IntegerGenerator();

    fooClass.callFunction(booClass.callSth().callAnotherFunction().lastCall());
    fooClass.callFunction(new Integer(intGenerator.getAnInt()).anotherCall());
}

```

Figure 3.7: Sample Java function - subject to source code retrieval and analysis, 3

First sample is very straightforward and its analysis and identification of functions should not be overly complex – however, our aim is to access the function body of called functions (functionCall(), anotherFunctionCall(), doSomething()). For that purpose the parser should not only identify called function names but also be aware of functions in super-classes and even super-classes of the super-class (very deep inheritance hierarchy). Proposed tool takes care of that and after identifying a function call declaration, it takes proper steps to identify the class to which a function declaration belongs.

Second sample demonstrates similar structure, however, function calls are bit more complex – the source code analyser has to additionally face overloading and identification of argument types. Again, source code analyser has to not only identify functions but also, precisely determine argument(s) type in order to match exact function call.

Third sample is the most complex of presented functions. Object Oriented programming allows for a lot of flexibility in terms of programming style and as a result source code analysers have to be able to work with such code. Proposed tools is also capable of identifying all of those functions which requires sequential analysis of given structure. Let's consider this line:

```
fooClass.callFunction(booClass.callSth().callAnotherFunction().lastCall());
```

In order to identify all 4 functions, first the class of booClass object should be identified. After that, the analyser can identify callSth() function and its return type – required to identify callAnotherFunction(). The same happens for callAnotherFunction() – its identification allows for identifying its return type and at the end, identification of lastCall() function. As a result, we can identify return type of lastCall() and in this way discover fooClass.callFunction(...). In practice, proposed tools do a 'backward search'. First, the analyser tries to identify callFunction(...). In order to do that, it tries to identify return type of lastCall(). During analysis of the last call, it goes backward (recursively) from lastCall(), to callAnotherFunction() and then to callSth() "asking" at each step for a return type. If the return type is identified, it identifies the function and gives the "answer" as a return in a recursive call.

Provided samples are not the ultimate cases for code that could be analysed – richness of Java programming language allows for much more complex

structures and declarations which proposed tool is also able to analyse. The problem could arise when argument types for function call cannot be precisely identified (for example, using concrete class while function is declared with interface).

3.4.4 User interface and code visualisation

The interface was designed in a way, that users can use the most essential features efficiently and in an intuitive way. The heart of the application is feature tree visualisation which allows navigating thru features with mouse and source code could be accessed with just few ‘clicks’.

Figure 3.8 shows the interface overview with opened project and modelled feature tree. The interface is concentrated on tabbed panes which allows for easily navigation between feature model, list of features and code of features. Having such layout, user can swiftly navigate from one view to another. Besides tree representation of the features, features are additionally displayed in a table layout 3.9 which additionally contains description of the feature – such view could be useful in case when there are many features and user wants to go through all of them.

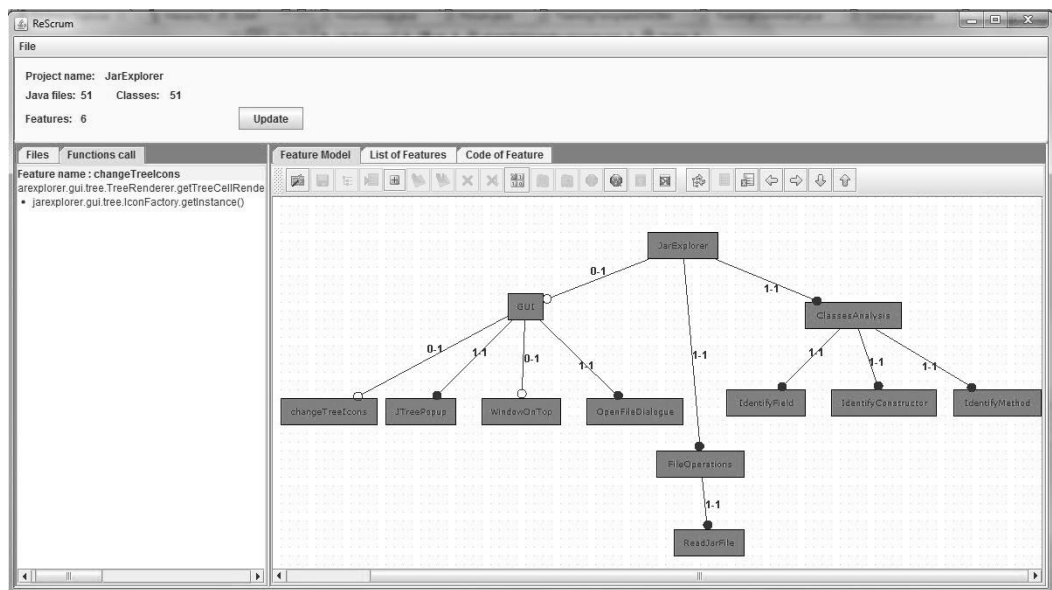


Figure 0.8: Main layout of the interface

Feature Model	List of Features	Code of Feature
Feature name	Feature desc	show code
JarExplorer	JarExplorer root - main method	Show feature
ReadJarFile	read jar file operations	Show feature
decompile	decompile sth	Show feature
changeTreeIcons	here we have code responsible for rendering tree and changing it's icons	Show feature
OpenFileDialogue	open file	Show feature
FileOperations	file operations	Show feature

Figure 0.9: Additional view with listed (discovered) features

When a single feature is selected via tree model by double clicking the feature or from the list of features, its representation is displayed in navigation view – “Functions call” presented in a figure 3.10. The feature is then presented as a hierarchy of function calls with multiple levels of depth so user can get familiar

with functions which are participating in an execution of a feature. Additionally, this view provides to the user information about the package and class of origin for each of the function as well as function signatures.

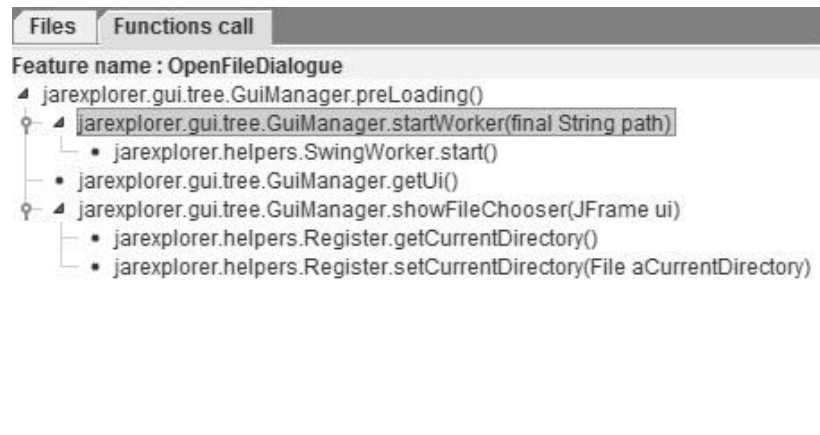


Figure 3.10: Functions call hierarchy for particular feature

The interface allows for easy access to the source code of the feature – via the navigation tree with function calls of each feature, user can access the source code just by clicking on appropriate function in the tree. Figure 3.11 demonstrates that after selection of `showFileChooser(JFrame ui)` function, the code of that feature is displayed and highlighted in “Code of Feature” table pane. That allows users to swiftly go through all functions participating in execution of particular feature.

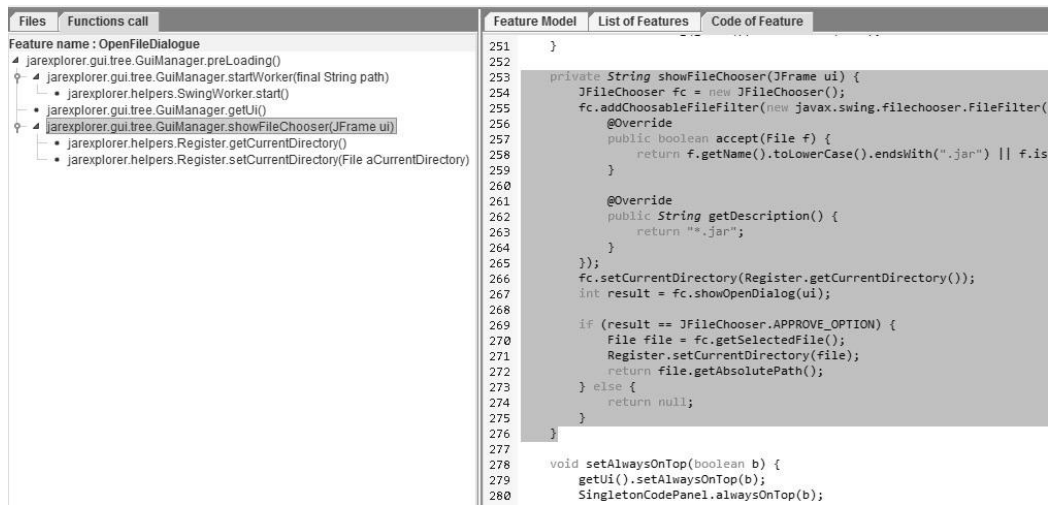


Figure 0.11: Interface with function call hierarchy and discovered code for particular function

Having such a simple interface, user can easily create project, feature model and access the source code of the features. This allows for easy interaction between developer and proposed tool which is one of the aims for proposed approach – to make it as lightweight and easy to use as possible.

3.4.5 Creating sample project

This section of the thesis demonstrates how to create a simple project according to proposed approach with use of delivered tool for retrieving source code for given features of a feature model. This simple, step-by-step description allows for better understanding of the approach as well as the tool. Proposed approach toward using the tool is not the only possible and some steps could be conducted in different order.

First step is to create a Java project with an IDE of a preference (Eclipse, NetBeans, etc) with included a Jar library into build path – the Jar library defines @Feature annotation so annotated code will compile without problems. After creation of the Java project, a reuse project with provided tool could be created.

In order to create a new project in the reuse tool, first, File->New project should be selected. After that user is asked to give a name of a project and select location of the source code that will be searched for annotations and potentially, reused. After confirming input data with “Create new project” user is asked to point location under which project’s files will be saved – now, the project is ready to use (see figure 3.12)

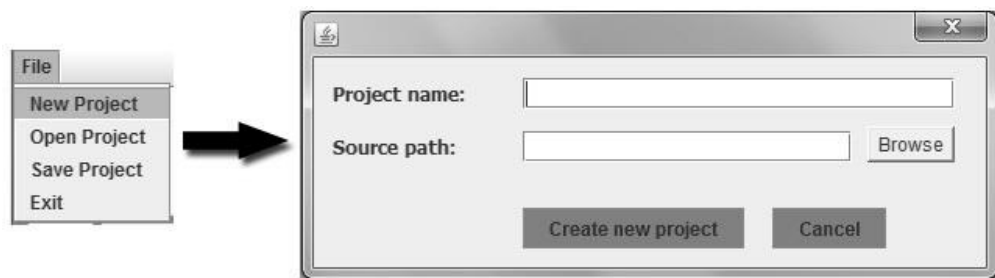


Figure 0.12: Creating new project

After having both projects started, next stage is to create a feature model. There are two possible approaches for that and first is favoured for better understanding of the project. The first approach is to create a whole feature model (feature tree) in advance and later on, annotate appropriate functions while writing a source code.

When new project is created, a design board for feature tree is empty and new features could be added and joined together to form a tree structure. Figure 3.13 shows two ways of adding a new feature – via a button located on a button bar or through popup menu (right click of a mouse). Figure 3.14 shows example of a tree model created with proposed tool. Such model could be also treated as additional high level documentation and description of a project.

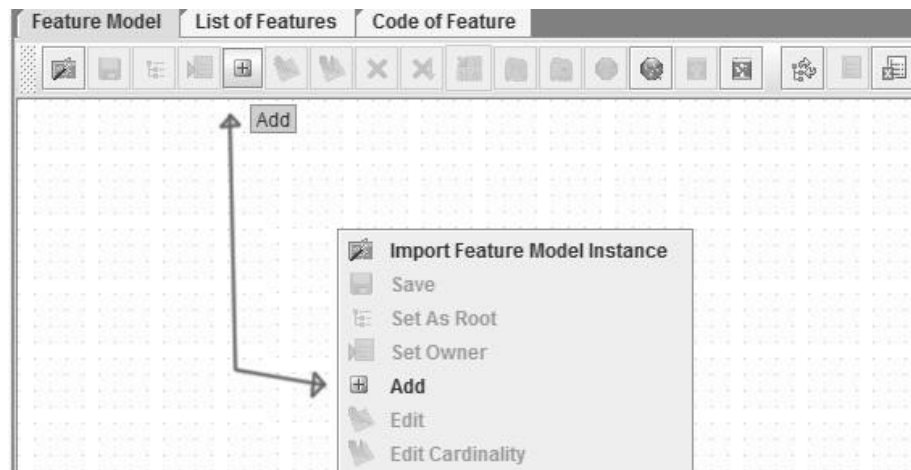


Figure 0.13: Interface for adding features to feature model

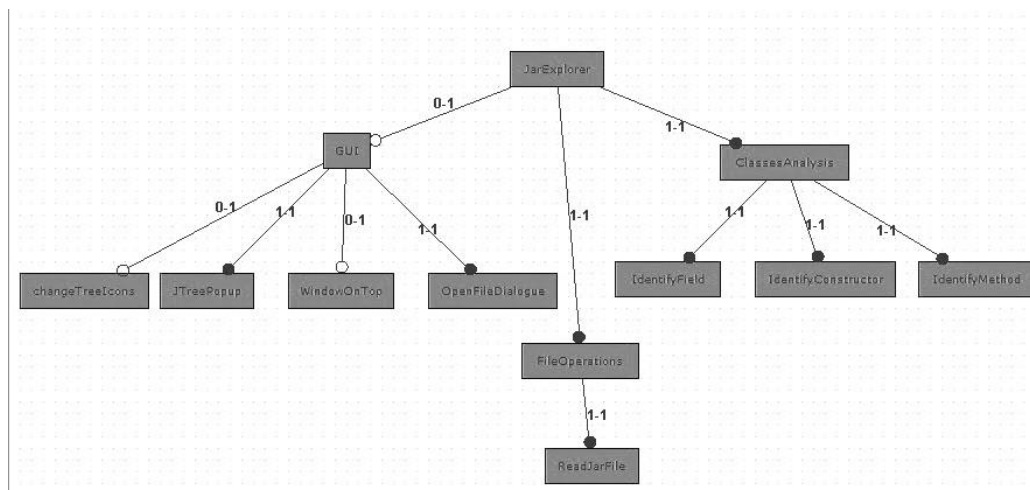
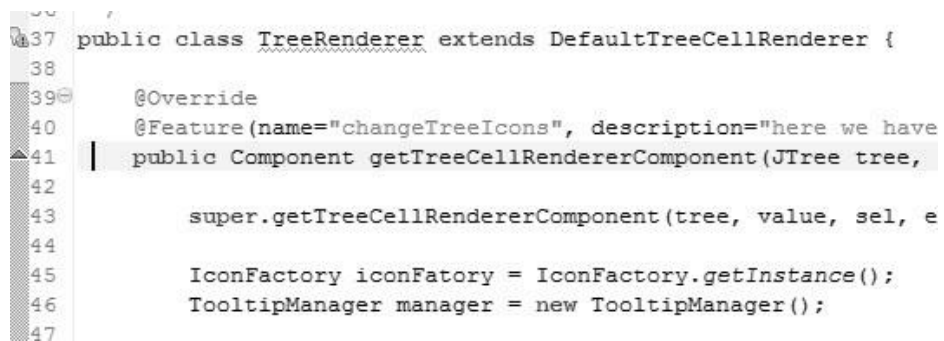


Figure 3.14: Sample feature model

Besides creating the feature tree in advance, it is also possible to first annotate features in a source code and they will be automatically discovered during creation, opening or update of the project. However, if it is decided to first annotate code and then discover such features, they will appear on a design board in unstructured and unconnected manner. Thus in order to have a tree model the

features have to be manually connected. In any case, when there is new feature in source code and we want to synchronise feature model with it, the project has to be reopened or an update action should be invoked via “update button” in a main layout.

Figure 3.15 shows piece of source code from an IDE with annotated function `getTreeCellRendererComponent`. After selecting a feature `changeTreeIcons` in a tree model (double click with a mouse) the feature function calls will be visible in a function call explorer on a left side of the layout (Functions call tab) – see figure 3.16. The functions call tab represents hierarchy of discovered functions calls within the code – only those functions of which source code is provided will be listed. By selecting any of given function name user can access the source code as it is presented on figure 3.17.

A screenshot of an IDE's source code editor. The code is in Java and shows a class `TreeRenderer` that extends `DefaultTreeCellRenderer`. The class has a method `getTreeCellRendererComponent` annotated with `@Feature(name="changeTreeIcons", description="here we have")`. The code is as follows:

```
37 public class TreeRenderer extends DefaultTreeCellRenderer {
38
39     @Override
40     @Feature(name="changeTreeIcons", description="here we have")
41     public Component getTreeCellRendererComponent(JTree tree,
42
43         super.getTreeCellRendererComponent(tree, value, sel, e
44
45         IconFactory iconFactory = IconFactory.getInstance();
46         TooltipManager manager = new TooltipManager();
47
```

Figure 0.15: @Feature annotated code in IDE



Figure 0.16: Function call tree with selected function to display in source code pane

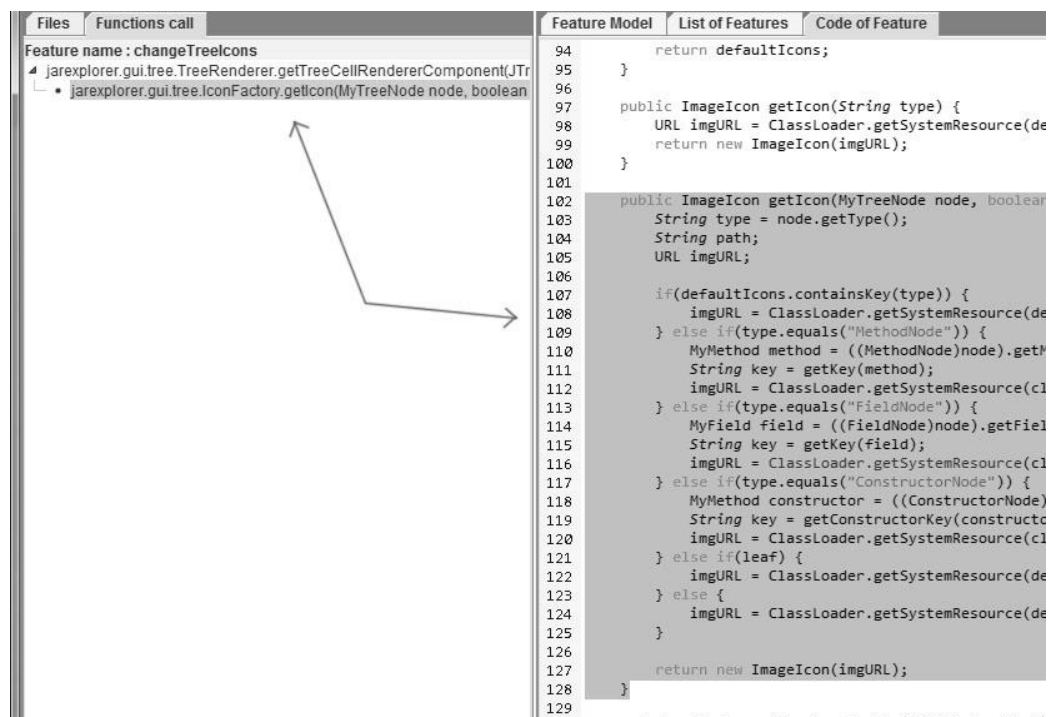


Figure 0.17: Interface with selected function in function call tree and displayed source code

CHAPTER 4

EVALUATION OF THE APPROACH

Evaluation of the approach is essential in order to understand deeper how beneficial its introduction would be for developers and organisation. By evaluating and testing, some question and ideas have also been generated which would allow to improve proposed approach to better match users' expectations on it. In order to evaluate the approach, appropriate tests have to be designed.

Tests for evaluating proposed approach were designed to answer several questions:

- **Does the approach work?**
This is a crucial question that have to be answered and will tell us if the approach meets its essential function or not – if the code retrieval works and can be retrieved easily.
- **What is a benefit of using the approach?**
When the approach is evaluated, it is checked if initial assumptions about possible benefits were correct or not. During evaluation, other potential benefits, not considered during approach design could be discovered and pointed out by testers.
- **How easy is it for developers to learn and understand how the approach works?**
When a new approach, framework or technology is introduced into a working environment, a learning curve is one of the considered aspects. The more complex or vague something is the more expensive and difficult it would be to implement in a working environment. Because of

that, during testing and evaluation such aspects are also addressed and considered.

- **What is developers view on the provided tool and approach?**
Besides testing the efficiency of the approach or a tool, also developers view on the approach is very important. It is essential that developers see benefits of using it and will cooperate both in assets generation as well as assets retrieval for making their work easier.

4.1 Test design and testing approach

Main aim of designing tests for proposed tool and approach was to reflect possibly precisely real environment and way in which both of them could be used. For that purpose, third party software - an open source application, was chosen as an object of test. The application was analysed and a feature model was created for it. Also, @Feature annotations were placed in a code to link feature model with source code. The feature model of that application is presented on a figure 4.1. It was decided that the object of tests should not be overly complex so participating in a test will not require great effort and time from voluntary testers.

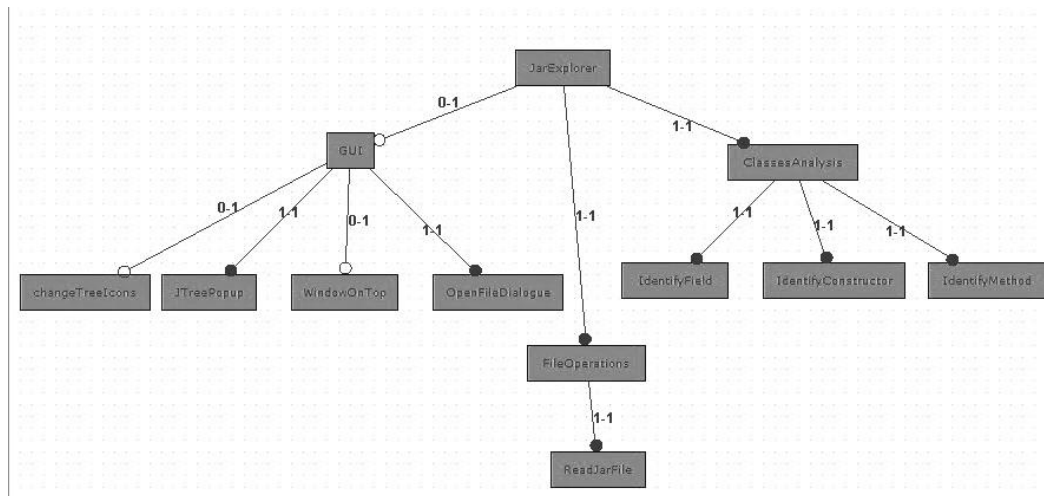


Figure 0.1: Sample feature model

Four Java developers participated in testing and evaluation. Each of the tests was carried out separately so testers did not have an advantage of understanding the tested code in advance. Before each test, it was explained how the approach works and how developers could benefit from it. After that, a tool for code reuse was presented with explanation how to use it and how to navigate through retrieved code. After that brief introduction, an application which source code will be retrieved during the test was presented. This open source application is an easy to use jar files browser and explorer – via simple interface, it allows to look inside into a jar of compiled classes and allows for navigating through it.

Explaining to testers how the tool for source retrieving and the approach works is essential to allow them working with it efficiently. Also, by demonstrating them the JarExplorer works will allow them to better understand what pieces of code they will be looking for.

In order to test the approach and provided tool, testers were asked to find in total source code of 4 features in provided source code of JarExplorer. Those features are related to application's GUI and most essential functions. Those features are:

1. Feature responsible for popping-up “open file dialog” for loading JAR file (feature 1)
2. Feature responsible for decompiling “.class” files from byte code to string (feature 2)
3. Feature responsible for keeping program window “on top” of other windows (feature 3)
4. Feature responsible for reading jar file and displaying it as a “tree” structure (feature 4)

What was measured during that test was how much time it takes for each developer to retrieve or identify source code of given feature. Each tester was asked to retrieve code of two features with use of provided tool for source code retrieval and two others without – tester had to navigate via source code and find those features without help of the tool.

In order to have more accurate results, different testers were asked to retrieve different pairs of features with help of provided tool and without it. For example, tester 1 was asked to retrieve source code of Feature 2 and 4 with provided tool for source code retrieval and Feature 1 and 3 with provided IDE

while tester 2 was asked to retrieve source code of Feature 1 and 3 with provided tool and Feature 2 and 4 with provided IDE.

4.1.1 Target group and environment

The tests were conducted on a group of 4 programmers (participants) familiar with Java programming language – all of them are currently a Master's students. One of them is currently working as a Java programmer (participant 1) while 3 others are familiar with Java programming language but currently are not working as programmers. Each test was carried out on a portable computer with Windows 7 operating system. Besides the tool for source code reuse, testers were allowed to choose one of two IDE on which they will be asked to retrieve code of a feature “by hand”. Those IDEs are: Eclipse and NetBeans which are the most popular non-commercial development environments. Testers were asked to assess their knowledge and experience with Java and other programming languages. The following information was provided:

- Participant 1 – working with Java for about 2 years, currently working as a Java programmer.
- Participant 2 – less than a year of experience with Java, familiar with other Object Oriented languages (about 2-3 years of .NET programming)
- Participant 3 – about a year of experience with Java, previous experience with other Object Oriented languages
- Participant 4 – less than a year of experience with Java programming

Testers were given the following task to complete:

Your task is to identify source code of 4 features within provided source code of a JarExplorer application. An example of such source code will be demonstrated to you in order to define “source code of a feature”. All of the features which you are asked to identify will be demonstrated to you in a working application, it means, it will be demonstrated how the feature works.

The features to identify:

1. *Feature responsible for popping-up “open file dialog” for loading JAR file.*
2. *Feature responsible for decompiling “.class” files from byte code to string*
3. *Feature responsible for keeping program window “on top” of other windows*
4. *Feature responsible for reading jar file and displaying it as a “tree” structure*

Besides, after conducting the tasks, participants were asked the following questions:

- How did you find usefulness of provided tool and approach?
- How would you improve provided tool in order to better perform your day-to-day programming duties with support of the tool?
- What are other benefits of using provided tool and approach?

4.1.2 Test results and participants’ feedback on the tool and approach

Participants were asked to discover a source code of four previously mentioned features. During the test, time was measured and results are presented in table 4.1. After analysing the results, it becomes clear that source code retrieval with provided tool becomes easier and takes less time. Average time for retrieving a source code of feature with provided tool was 50 seconds while “by hand” (with IDE) it took on average 3 minutes and 28 seconds.

Table 4.1: Time required for retrieval and identification of a source code for particular feature. Measured for different participants with and without provided tool

	Time for identifying source code a features 1,2,3,4			
	Retrieval with tool		Retrieval "by hand" (with IDE)	
	Number of a feature (time)		Number of a feature (time)	
participant 1	2 (45sec)	4 (1min)	1 (2min)	3 (3min)
participant 2	1 (1min 10sec)	3 (40sec)	2 (6min, 30sec)	4 (2min, 5sec)
participant 3	2 (35sec)	4 (50sec)	1 (4min 10sec)	3 (1min 15sec)
participant 4	1 (1min 15sec)	3 (25sec)	2 (5min 35sec)	4 (3min 15sec)

Beside tests, participants provided also very valuable information of possible improvements and usage of proposed tool and approach. All of suggested suggestions were about improving the tool by adding additional functionalities and making operation of it easier – improving interface. Participants proposed following improvements of the tool:

- Adding possibility of folding/unfolding nodes of feature tree so in case of very large trees, it would be easier to read it and find appropriate feature.
- Adding tooltips to the nodes of the feature model tree with additional information such as feature description or function names constituting the feature.
- Providing given tool as a plug-in for Eclipse IDE so operation of it would be easier to operate on a source code and feature model at the same time – i.e. directly modifying the code after accessing it via feature model. However, such functionality was not considered for given approach, because primary use of it was designed in order to ‘retrieve’ and ‘reuse’ source code rather than operate on an application and maintain it.

What is also very interesting, participants found and proposed a new way of using provided tool and approach. One of them have suggested, that such tool would be very good for source code maintenance, because when new programmer is assigned to maintain a source code he/she can easily identify particular features within code. Such suggestion is in parallel with statement from the thesis, that such feature model would be an additional high-level documentation of the source code. Also, another participant of the test suggested that proposed tool allows for faster and better comprehension of a source code.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

Proposed work is just a cornerstone for future research and improvements of the proposed approach and a tool for source code retrieval. During tests and evaluation it was demonstrated that both approach and the tool allowed for retrieving of source code that developers were asked to find. However, in order to make more accurate conclusions and judgements both approach and the tool should be evaluated in a real production environment on more than one project – in order to first generate assets and then to use those assets or simply, reuse written code.

The approach could be also called ‘seamless assets generator’ because in contrary to other software reuse methodologies, it does not require to generate or write assets in advance before starting any project. Because assets are generated while working on ‘normal’ projects and applications, thus no extra workforce is required to participate in software reuse processes. It is big advance over other methodologies because it reduces risk of introducing approach and reduces costs of using it.

Proposed system (approach + tool) is easy and cheap to introduce into production environment thus at almost no-cost could be tried and evaluated by software developers. This fact is very important because high costs of

introduction of something new into a company, could be a major obstacle in its implementation. Also, when a company decides to use given system, it can be abandoned at anytime without damage or any inconvenience to developers. Because entry and exit costs of using the system are low, there is a chance that development teams decide to try provided system.

The system is currently working, but still it is in an early stage of development thus there is high chance that if a company would decide to use it, many ideas will come into developers' minds about possible improvements and extensions. More experienced software architects and engineers who participated in variety of different projects and more than once faced issues of source code reuse, would probably contribute the most and their ideas could help to make such system 'development and enterprise forged'.

At the current stage, the tool for code retrieval is a separate application but for further work, it would be more convenience to have such tool integrated into such IDE as Eclipse or NetBeans. By integrating the tool into IDE, developer will not be required to start and operate another application while working on a project – it would make his/her environment more users friendly and comfortable. Also, by integrating such tool into IDE, developer would be able to navigate through retrieved code and code he/she is working at a time easier – just by switching between tabs. Both of mentioned IDEs are plug-in friendly and if significant amount of time is invested, experienced Eclipse or NetBeans contributors could create such extensions.

Proposed approach is supposed to be scalable and work equally well with small, medium and large number of former projects stored as company assets. However, at current stage it could be more difficult to work with large number of projects because for retrieving a feature code, developer would have to traverse

large number of projects. He/she would be lucky if in advance he/she would know in which projects to look for a feature – for example, within a small (5-10) set of projects. Now, let's imagine that developer does not have such knowledge and has to look through variety of projects to find a desired feature. In such case, usage of approach would become less beneficial. In order to prevent such situation and in general, improve the approach a simple extension of it could be introduced. One of such extension is storing features names and descriptions in a database with search interface. In such scenario, when developer would be looking for a feature, he/she could first do a keyword search in a database. That would allow for identifying a project or small set of related projects. Another possible improvement is introduction of ontology – related projects could be after completion linked with each other in an ontological way. Such ontology could play a role as a roadmap of all projects and would allow developers to access related projects and locate features they are looking for.

Another possible improvement of provided approach is to track which features are reused in which projects. For example, if a Feature A originally implemented in Project A was later reused in Project X, Y and Z it would be beneficial to keep track of such actions for various reasons. Let's consider that during Project A operations it was discovered that Feature A has a serious bug which occurs in really rare situations and just in special conditions thus it was hard to discover the bug earlier. If we have information where the feature was reused, then we can check if affected code was also reused or not. Another possible benefit of keeping track of features reuse is to identify the features which are most commonly used. If we can track them, then proposed approach could be connected with other reuse based software methodologies and appropriate components or libraries could be implemented. Such tracking should not be difficult to implement and could be done in at least two ways. First, there

could be a database with web interface where developers are submitting facts of reusing particular feature in particular project with some details that allow identifying where exactly that code was reused. Another possibility is to introduce another annotation (for example, `@Reused (project = "projectName", feature = "originalFeature")`) which indicates where the code was reused. This method would allow for using automatic research tools for analysing which features were reused and where, which has such benefit that developer him/herself would not be obligated to use any web form to submit fact of reusing feature. By introducing such monitoring tools or techniques, it would be possible to assess in a long time prospect, how effective the approach is. This would allow objectively confirming or rejecting benefits of using the approach.

At the time of writing this thesis, there are no standard tools or files formats for modelling and representing feature models. However, if feature modelling will gain enough popularity then for sure we can expect that such tools will emerge on a market. Having professional tools for modelling would make modelling process easier and also, more complex models could be created and maintained. In a case when such tools become available one day, it would be reasonable to make proposed tool compatible with such file formats/standards so models created with dedicated feature modelling tools could be read by tool for retrieving source code for features. In this way, a lead developer who is responsible for maintaining feature model could benefit from using fully dedicated and robust tool for feature modelling while developer at the same time would be able to read the feature model into reuse tool and retrieve code of particular features.

At the current stage, provided tool for source code reuse is working just with Java code and as future work, the tool could be extended to work with other popular programming languages such as C#, C++, Ruby or Python. It would

require using different language parsers for source code and rewriting some of the modules for code retrieval. It could be challenging but still possible to have an all-in-one tool for source code reuse which supports mentioned programming languages.

BIBLIOGRAPHY

- [1] javaparser <http://code.google.com/p/javaparser/>, Last access date: 15 March 2011.
- [2] S. Apel, C. Kaestner, and C. Lengauer. "Research challenges in the tension between features and services". In *Proceedings of the 2nd international workshop on Systems development in SOA environments, SDSOA '08*, pages 53- 58, New York, NY, USA, 2008. ACM.
- [3] S. Apel and C. Kstner. "An overview of feature-oriented software development". *Journal of Object Technology*, vol. 8, no. 5, July-August 2009, pp. 49-84
- [4] K. Berg, J. Bishop, and D. Muthig. "Tracing software product line variability: from problem to solution space". In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, SAICSIT '05*. South African Institute for Computer Scientists and Information technologists, 2005.
- [5] T. J. Biggerstaff, B. G. Mitbender, and D. Webster. "The concept assignment problem in program understanding". 15th International Conference on Software Engineering, Baltimore, MD , USA, May 1993.
- [6] D. Binkley. "Source code analysis: A road map". Future of Software Engineering (FOSE '07), 2007, pp. 115-30, May 23-25, 2007, Minneapolis, MN, USA,
- [7] G. Canfora, A. Cimitile, and M. Munro. "Re2: Reverse-engineering and reuse re-engineering". *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 6, issue 2, 1994.
- [8] D. Clarke and J. Proenca. "Towards a theory of views for feature models". FMSPLE, Jeju Island, South Korea, 14 September 2010, 2010.
- [9] J. M. Conejero and J. Hernndez. "Analysis of crosscutting features in software product lines". *Proceedings of the 13th international workshop on Early Aspects*, New York, NY, USA, 2008.

- [10] K. Czarnecki and U. Eisenecker. “Generative programming: Methods, tools, and applications”. 2000.
- [11] R. M. Dijkman, D. A. C. Quartel, and M. V. Sinderen. “Consistency in multi-viewpoint design of enterprise information systems”. *Information & Software Technology*, Volume 50, Issue 7-8, 2008.
- [12] S. Ferber, J. Haag, and J. Savolainen. “Feature interaction and dependencies: Modeling features for reengineering a legacy product line”. *In Software Product Lines*, Springer Berlin, 2002.
- [13] O. Greevy, S. Ducasse, and T. Grba. “Analyzing software evolution through feature views”. *Journal of Software Maintenance and Evolution: Research and Practice*, 2006.
- [14] M. Griss, J. Favaro, and M. d’Alessandro. “Integrating feature modeling with the RESB”. In International Conference on Software Reuse, 1998.
- [15] M. Harman, N. Gold, R. M. Hierons, and D. Binkley. “Code extraction algorithms which unify slicing and concept assignment”. In *Working Conference on Reverse Engineering*, 2002.
- [16] P. Heymans, P. –Y. Schobbens, J. christophe Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. “Evaluating formal properties of feature diagram languages”. *IET Software*, Volume 2, Issue 3, 2008.
- [17] J. Highsmith. “Agile Software Development Ecosystems”. Addison-Wesley Professional, 2002.
- [18] K. Kang, K. Lee, J. Lee, and S. Kim. “Feature-oriented product line software engineering: Principles and guidelines”. 2003.
- [19] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. “Feature-oriented domain analysis (FODA) feasibility study”. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
- [20] A. S. Karatas, A. H. Dogru, H. Oguztuzun, and M. Tolun. “Using context information for staged configuration of feature models”. *In Transformative Systems Conference: SDPS 2010*, 2010.
- [21] C. W. Krueger. “Software reuse”. *ACM Computing Surveys*, Volume 24, Issue 2, June 1992.

- [22] C. Larman. "Agile and Iterative Development: A Manager's Guide". Addison-Wesley Professional, 2003.
- [23] H. Mei, W. Zhang, and H. Zhao. "A metamodel for modelling system features and their refinement, constraint and interaction relationships". *Software & System Modeling*, Volume 5, 2006.
- [24] H. Mili, A. Mili, S. Yacoub, and E. Addy. "Reuse-Based Software Engineering: Techniques, Organizations, and Controls". Wiley-Interscience, New York, NY, USA, 2001.
- [25] I. Pashov, M. Riebisch, and I. Philippow. "Supporting architectural restructuring by analyzing feature models". In *Conference on Software Maintenance and Reengineering*, pages 25- 36, 2004.
- [26] K. Pohl, G. Böckle, and F. J. v. d. Linden. "Software Product Line Engineering: Foundations, Principles and Techniques". Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [27] C. Prehofer. "Feature-oriented programming: A fresh look at objects". In *European Conference on Object-Oriented Programming*, 1997.
- [28] C. A. Szyperski. "Component software - beyond object-oriented programming". Addison-Wesley-Longman, 1998.
- [29] O. Üçtepe. "Utilization of feature modeling in axiomatic design". Master's thesis, Middle East Technical University, 2008.
- [30] M. Weiser. "Program slicing". *IEEE Transactions on Software Engineering*, 1984.
- [31] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. "A comparison of methods for locating features in legacy software". *Journal of Systems and Software*, Volume 65, 2003.
- [32] W. E. Wong, S. S. Gokhale, and J. R. Horgan. "Quantifying the closeness between program components and features". *Journal of Systems and Software*, Volume 54, 2000
- [33] Y. Yu, J. Mylopoulos, A. Lapouchnian, and S. Liaskos. "From stakeholder goals to high-variability software designs" 2005.