jamk.fi

Assisting Software Quality Assurance by Change Impact Analysis

A tool-driven Approach

Iiro Hietala

Master’s thesis
December 2014

Master's Degree programme in Information Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Hietala, Iiro	Master's thesis	3.12.2014
		Language of publication: English
	70	Permission for web publication: Granted

Title of publication

Assisting Software Quality Assurance by Change Impact Analysis

A tool-driven Approach

Degree programme

Master's Degree in Information Technology

Tutor(s)

Peltomäki, Juha

Assigned by

Codecenter Oy

Abstract

The primary goal of this thesis was to research the previous software change impact analysis research and the availability of impact analysis tools targeted for quality assurance, i.e. tools that analyse the Actual Impact Set. As no such tools were readily available, an open-source tool called *J-Ace* was designed and implemented as part of this Thesis.

The design consisted of laying out requirements for the tool and the selection of the technologies that were used to implement the tool. The implementation succeeded in fulfilling the primary requirements that were laid out for the tool.

Two case-studies were performed using the implemented tool to assess the fulfillment of requirements that were laid. The first case-study demonstrates the analysing of changes made to the source code of the tool itself. This was used to test out the functionality of the software and the impact scoring of the change-sets.

The second case-study demonstrates the ability of analysing an open-source *Enterprise Content Management* system called *Alfresco*. The issue-tracker of Alfresco was studied to pick issues that were analysed using the tool. Then the results from the tool were compared to the issue description to identify whether the quality assurance team could have tested out the correct parts of the system based only on the issue description.

The results of the case-studies gave a positive outlook of the J-Ace features and encourage for further development of the tool.

Keywords/tags

Impact analysis, Java EE, Quality Assurance, Tool

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Hietala, Iiro	Opinnäytetyö	3.12.2014
	Sivumäärä 70	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: Kyllä

Työn nimi

Assisting Software Quality Assurance by Change Impact Analysis

A tool-driven Approach

Koulutusohjelma

Master's Degree in Information Technology

Työn ohjaaja(t)

Juha Peltomäki

Toimeksiantaja(t)

Codecenter Oy

Tiivistelmä

Opinnäytetyön päätavoite oli koota aiempien tutkimuksien tuloksia liittyen ohjelmistojen muutosvaikutusanalyysiin ja löytää työkaluja laadunvarmistustiimien käyttöön, jotka analysoivat sovellukseen tehtyjä todellisia muutoksia. Tutkimuksen mukaan olemassa olevia työkaluja tähän tarkoitukseen ei löytynyt. Uusi työkalu nimeltä '*J-Ace*' suunniteltiin ja kehitettiin osana opinnäytetyötä.

Suunnittelutyössä suoritettiin vaatimusmäärittely työkalulle, sekä valittiin teknologiat, joilla työkalu toteutetaan. Toteutustyössä onnistuttiin täyttämään päävaatimukset, jotka työkalulle oli asetettu.

Työssä suoritettiin kaksi tapaustutkimusta, joiden avulla tutkittiin työkalulle asetettujen vaatimuksien täyttymistä. Ensimmäinen tapaustutkimus tutki itse työkaluun tehtyjen muutoksien analysointia pyrkimyksenä kartoittaa työkalulle asetettujen vaatimuksien täyttymistä sekä vaikutuspisteytyslaskennan toimivuutta.

Toinen tapaustutkimus tehtiin avoimen lähdekoodin sisällönhallintajärjestelmälle nimeltään *"Alfresco".* Alfresco-projektin tikettienhallintajärjestelmästä valittiin tikettejä, jotka analysoitiin työkalun avulla. Tarkoituksena oli vertailla tikettienhallintajärjestelmään syötettyjen tietojen totuudenmukaisuutta itse toteutettuihin muutoksiin nähden. Tämän kautta voidaan arvioida, pystyisikö laadunvarmistustiimi testaamaan sovelluksen vain tikettienhallintajärjestelmän pohjalta.

Tapaustutkimuksien tulokset olivat positiivisia, ja työkalu täytti sille asetetut päätason vaatimukset. Työkalun kehitystä tullaan jatkamaan tutkimustuloksiin pohjautuen.

Avainsanat (asiasanat)

Muutostenhallinta, Muutostenvaikutusanalyysi, Java EE, Työkalu, Laadunvarmistus

Muut tiedot

Contents

Figures

Tables

Terms and Acronyms

Term or Abbreviation	Description
AIS	See Actual Impact Set
Actual Impact Set	The Actual Impact Set defines the actual software life-cycle objects that have been changed in a change set
Black Box Testing	Testing of a software based on inputs- and outputs of the software without the knowledge of the internal workings of the software
CIA	See Change Impact Analysis
Change Impact Analysis	Change Impact Analysis is a technique for identifying the effects of a change or estimating necessary components to be modified to achieve a change
Change Set	A grouped set of files that indicate a change in a version control system
Conceptual coupling	A dependency or relation between SLOs
Enterprise Software	The software of the Enterprise Application that provides benefits to a certain business area of an enterprise. In this document this mainly refers to a common J2EE application with multiple integrations to other systems.
False-negative Impact Set	The software life-cycle objects that are not in the estimated impact set but are contained in the actual impact set. i.e. the files that were not estimated to be changed but were changed.
False-positive Impact Set	The software life-cycle objects that are included in the estimated impact set but are not contained in the actual impact set. I.e. the files that were estimated to be changed but were not changed.
Java EE	Java Enterprise Edition
SLO	See Software Life-cycle Object
Software Life-cycle Object	A source code file, a resource, a properties-file, a configuration file or similar that affects the functionality of the software system
VCS	See Version Control System
Version Control System	A system that manages the revisions of documents such as source code files
White Box Testing	Testing of a software with knowledge if the internal structure of the software.

1 Acknowledgements

I sincerely thank my spouse Marjo Janhonen for supporting me through this endeavour and giving birth to our two lovely kids and taking care of them when I was unavailable during writing the J-Ace tool and this thesis.

I also want to thank Juha Peltomäki for guiding me in the writing process.

2 Introduction

2.1 Background

In the field of software development the delivered software system tends to change often. The changes may be roughly divided into in four categories: the *perfective* changes that aim to increase business value of the software by introducing new functionality and to improve existing features, the *corrective* changes that provide corrections to erroneous functionality or bug corrections, the *adaptive* changes that adapt the software to function in new circumstances or environments. and finally there are the *protective* changes that are invisible to end-user, however they rather aim to improve the system through re-factoring and other proactive changes (Rajlich 2012, 69-73).

According to Rajlich (2012, 82) surveys indicate that of all the software engineering work up to 80 percent is performed on software changes. Glass (2001) argues that maintenance costs of a software system consume about 60 percent of software costs on average, which means that the software is constantly changing through its life-cycle.

Especially in complex enterprise systems the developer is implementing the changes to one part of the system may cause a ripple-effect that affects other parts of the system in unintended manner (Rajlich, 2012, 106-107).

After implementing a change the system functionality may be tested automatically through a set of automated unit and integration tests. Yet, it is not realistically possible to achieve 100 percent coverage of all code paths with all the possible data variations. Manual acceptance testing needs to be performed to ensure that the implemented changes meet the actual business requirements. All the affected features must also be tested for regression.

The role of Quality Assurance (QA) is to verify the implementation and test the system for any regression. As exhaustive system testing may not be performed after each change there clearly is a need to identify only the affected parts of the system related to the change for the QA to test. When manual testing is planned and performed on the change set the information about all the changed parts of the system ultimately depend on the communication between the developer and tester and their expertise. Tools such as *JIRA*, *Stash,*

GitLab etc. may also be used to manually keep track of the changes, however these tools do not provide high-level visibility of the changed features of the system. If either a developer fails to detect the affected system components at commit-time or fails to deliver the information of the changed features to the quality assurance there is a high risk of defects being introduced into the system.

This study researches the possibility of decreasing the chance of regression by introducing a software Change Impact Analysis (IA) technique to the process. The research focuses on existing techniques and tools for the aforementioned purpose and proposes a new tool to perform IA based on data-mining of changes made into software repository. The purpose of the tool is to help alleviating the problems faced in real-life Java Enterprise Edition (Java EE) projects regarding the quality assurance challenges of where a limited amount of testing personnel are available and tight time-constraints apply.

The Java EE applications tend to be complex, have multiple integrations to other systems and usually are business-critical for the company using them (Farley et al., 2005, 4-5). All the features in the system cannot be functionally tested every time a software change is committed. In complex projects the chance of regression in such circumstances is high. To lessen the chance of regression the impact of code changes must be analysed. The thesis proposes a tool for efficiently tracing the software changes in Java EE projects.

2.2 Scope and objectives of this study

This study focuses on dependence based IA techniques and strictly on analysis of Actual Impact Set (AIS) (Bohner, 1996, 38), i.e. to the software changes that have already been performed. The change sets are data-mined from the version control system. In agile projects changes to several parts of the system are done often. The previously implemented requirements are not necessarily managed for changes as the features are done in iterative manner where each iteration holds a particular set of user stories to be fulfilled at implementation level. Dependencies between the user stories are usually managed only when user stories are being written for new features. When an existing set of features need to be changed the old user stories may not be efficiently traced

back unless efficient requirements and change management processes are followed (Biggelaar, 2014, 23). The software is being constantly changed due to changing requirements. In such project environment the impact analysis may only be performed efficiently after the changes have been implemented and thus traceability based IA techniques are out of scope.

2.3 Research Methodology

As introduced a need exists for a tool or a process to catch regression bugs earlier. A technique for performing Actual Impact Set analysis is needed. For the purpose of creation of a new Change Impact Analysis technique the Design Science Research methodology (DSR) was selected for this study.

Peffers, Tuunanen, Gengler, Rossi, Hui, Virtanen & Bragge (2006, 89-92) gathered a synthesis of existing publications about DSR methodologies for Information Systems research and proposed the following model for producing and presenting information for Information Systems research:

1. Problem Identification and Motivation
    - Define the specific research problem accurately to justify the development of an artifactual solution to the problem
    - Motivate readers to understand the subject and the reasoning of the researcher and to accept the importance of the solution
2. Objectives of a solution
    - Derive the objectives from the problem description based on current knowledge of the state of the problems and possible solutions
3. Design and Development
    - Define the desired functionality and architecture of the artifact based on objectives and create the artifact
4. Demonstration
    - Use the artifact to demonstrate its purposefulness to solve the problem. This could be an experimentation, a simulation, a case

study, a proof or similar activity

5. Evaluation
    - Evaluate the results of the demonstration by comparing them against the set objectives.
    - Decide based on results if plausible whether to improve the artifact by back-tracking to step 3 or to move on to step 6 to communicate the facts for future researches to address
6. Communication
    - Communicate the research, the problem and its importance, the solution, and results to suitable audience.

2.4 Structure of Thesis

The structure of this thesis generally follows the DSR model proposed by Peffers et al. (2006, 89-92).

- In Chapter 3 the previous research about the subject is investigated and introduced. The feasibility of the previous approaches is determined in contrast of the problem defined in chapter 4.
- In Chapter 4 the problem is identified and the basics of the quality assurance processes and Java EE application architecture are introduced within the scope of the problem.
- In Chapter 5 the objectives for solution are defined.
- In Chapter 6 the focus is in design and development of the Change Impact Analysis tool.
- In Chapters 7 and 8 the tool is demonstrated by using two case-studies of two existing open-source Java EE applications
- In Chapter 9 the results of the demonstration against the defined objectives are evaluated.

- The chapter 10 contains the conclusions of this study.

In this thesis the focus is kept in the context of the original problem.

3 Background and related research

The background research consists of existing IA techniques from published papers and literature. The scope of the research is directed on IA techniques that concentrate on producing tools to automate the IA process and especially on tools that focus on data-mining of software repositories.

3.1 Theory of Change Impact Analysis

There are two commonly identified methods to approach impact analysis. The *Traceability* based IA concentrates on tracing the dependencies between artifacts such as requirements, design documents, and source code files (Lehnert, 2011, 2).

The *Dependence* based IA concentrates on analysing the changes and ripple effects thorough the software system initiated by a software change (Li et al., 2011, Chapter 1.2). As stated in Chapter 2.2 this study focuses on dependence based IA techniques.

When a software change to an existing system or a feature is required, there are usually multiple files affected that define the classes, resources, configuration files, database tables and other files. These file types may be called Software Life-cycle Objects (SLO) (Bohner & Arnold, 1996, 42; Bohner 2002, 177).

The SLOs may have dependencies between each other thus causing a need for change in multiple parts of the software. The process begins with the assessment of the affected SLOs. The initial assessment produces the set of SLOs that the change will affect directly. Bohner (1996, 38) calls this as *Starting Impact Set* (SIS). Other terms may also be found in literature and papers, for example Rajlich (2012, 106) calls this the *initial impact set*. In this thesis the terminology introduced by Bohner is used.

However, there are usually dependencies and interactions from the SLOs defined in the SIS to the other SLOs that are not contained in SIS. These SLOs must be identified by analysing the dependencies and interactions. The

dependant set of SLOs estimated to be affected by the change to SIS is called the *Estimated Impact Set (EIS). (*Bohner, 1996, 38). These are also referred to as *secondary modifications (Rajlich, 2012, 106)*. The estimated impact set also includes the starting impact set.

When the software change is performed to the SLOs the affected modules form the *Actual Impact Set* (AIS) (Bohner & Arnold, 1996, 38). If the EIS is equal to AIS then the impact analysis has been performed perfectly.

It should also be noted that AIS is not usually unique since there are multiple approaches how to perform the software change (Bohner 1996, 38; Li, Sun, Leung & Zhang, 2012, Chapters 2 – 2.2).

In their survey of existing IA techniques Li, Sun, Leung and Zhang *(2012, Chapter 2.2.)* also identify the *False Positive Impact Set* (FPIS) and *False Negative Impact Set (FNIS)*. The FPIS contains the SLOs that have been identified into EIS but that are not contained in AIS. These are the modules that did not change as estimated. The FNIS contains the SLOs that have not been identified in EIS but are contained in the AIS. These represent the modules that were changed but that could not be identified in the estimation phase.

3.2 Impact analysis process

The process of change impact analysis may be *informal* or *formal*. By the *informal way* the change locations (e.g. SLOs) are determined by the developers without an iterative process and are based solely on the expertise of the developers and their knowledge of the system at hand. The analysis may be done without formally addressing it as impact analysis, nevertheless it is performed (Lindvall, Sandahl, 2003, 26-27).

The *formal way* is to perform impact analysis in iterative and incremental manner. Figure 1 illustrates this process.

1. The SIS is analysed and identified for a change request as a manual process based mainly in the expertise of the developer (Johnson, 2005, 39).

2. After the SIS is identified the impact analysis is performed to find dependencies and interactions between the SLOs contained in SIS and other SLOs. This set of SLOs form the EIS. Also the further dependencies and interactions of the SLOs contained in the EIS are analysed and added to the EIS enlarging the EIS after each analysis iteration. After no more dependencies are found the EIS is complete.
3. The changes are performed to the system. The set of SLOs that were actually affected form the AIS.
4. The SLOs that were changed and are contained in AIS but that were not in EIS form the FNIS.
5. The SLOs that were not changed and are not contained in AIS but that were included in EIS form the FPIS.
6. If EIS = AIS, the change impact analysis process was perfect and identified all the changes to be made beforehand.

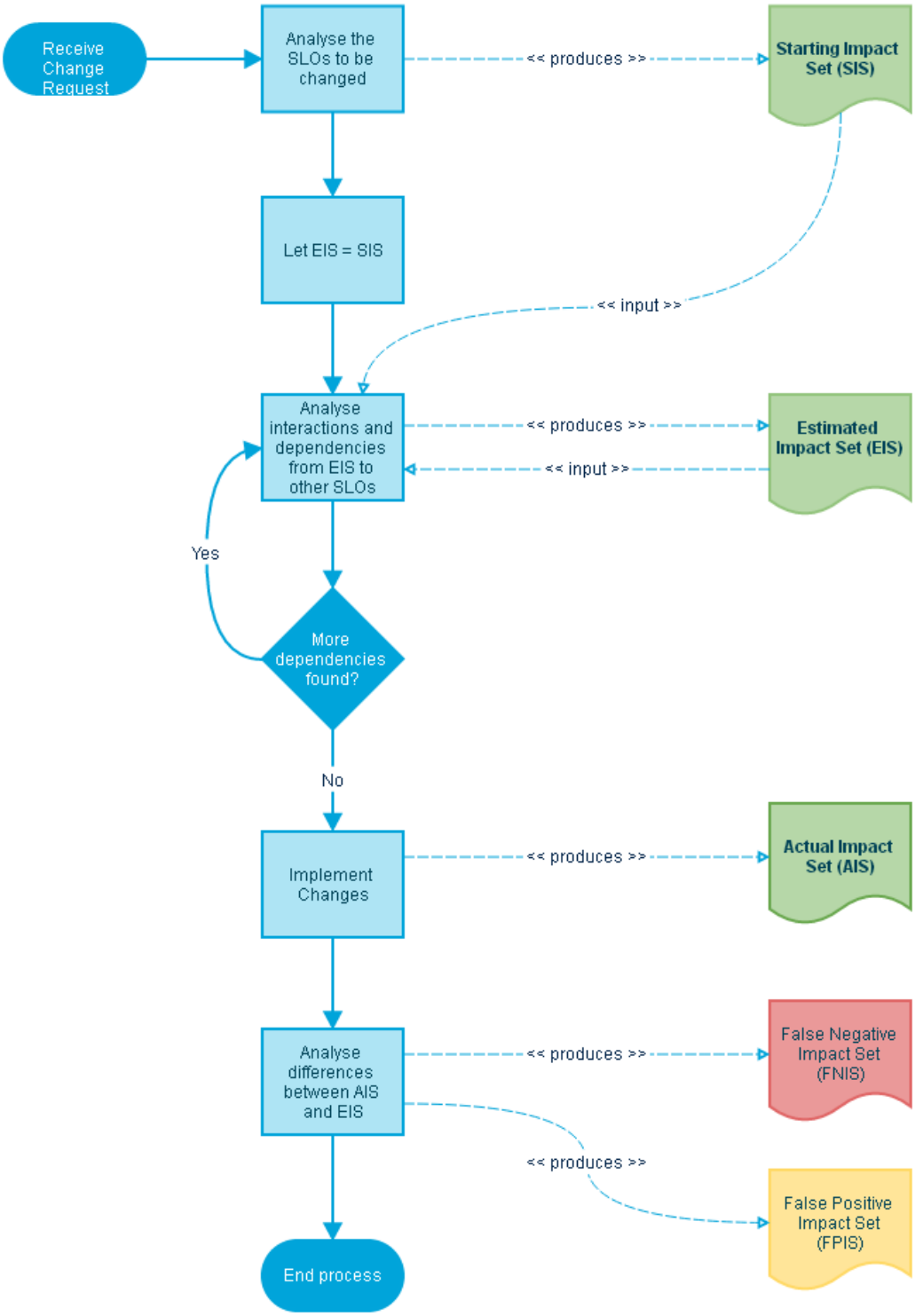


Figure 1: Process of impact analysis

Rajlich (2012, 112-113) proposes that during the impact analysis process the SLOs are marked with one of the following marks:

- *Blank*, meaning that SLO was not inspected nor scheduled for inspection
- *Changed*, indicating that SLO will be impacted by the change and is part of EIS
- *Unchanged*, marking the SLO as inspected and outcome of analysis is that SLO will not be affected by change
- *Next*, scheduling the SLO for inspection during the impact analysis
- *Propagating*, which indicates that the SLO will not be impacted by the change directly but the dependencies of this SLO may change, i.e. the change may propagate to dependant SLOs

Initially the SLOs are marked as *blank*. Then the process of impact analysis begins and SIS is identified as defined in the Figure 1. The files in SIS are marked as *changed*. Then all the dependant SLOs are marked as *next* identifying the initial EIS. After this the analysis process continues iteratively by analysing the EIS SLOs and their dependencies and marking them and their dependencies with a proper marking. When *propagating* mark is used then the dependencies of the propagating SLO are marked with *next*. The process continues until no more SLOs with *next* marking may be found (Rajlich, 2012, 112-116).

3.3 Granularity of SLOs

The software life-cycle object may be anything from a text-file, properties-file, class file, XML configuration file to a database change script or any other meaningful resource that affects the functionality of the software. The granularity is meaningful when analysing the SLOs. The granularity in scope of impact analysis indicates the level on analysis. When considering the source files a fine level of granularity may indicate the class members and methods. A coarser level may indicate inner classes, the whole classes or whole files. (Li et al, 2012, Chapter 6.2).

According to Petrenko & Rajlich (2009, Chapters 3.3 and 5) the chosen granularity level affects the precision of the analysis. The precision is higher as as the level of granularity is finer.

3.4 Survey of existing IA techniques

A research by Li et al. (2012, Chapter 4) was conducted to survey existing IA techniques and publications from years between 1997 and 2010. The sources were four major digital databases: *ACM Digital Library, IEEE Computer Society Digital Library, Science@Direct* and *Springer Link*. The research excluded technical reports and other non-verifiable material as according to Li et al. their quality cannot be guaranteed (ibid. Chapter 3). The initial amount of publications found was 2357 and by exclusion of irrelevant papers such as that do not focus on dependency based IA techniques the final set to study was reduced to 30 publications. Of these publications 9 provide techniques that are based on data mining of software repositories (ibid. Chapter 4). This study reviews the purposefulness of these 9 techniques to the problem defined in Chapter 4.

The trends in the survey show that IA techniques based on data-mining of software repositories constitute a rising trend when compared to the traditional dependency- and traceability based IA techniques (ibid. Chapter 4).

Based on the paper (ibid. Chapter 4) the most promising and interesting publications in relation to this study are listed in Table 1.

Table 1: Existing publications and IA techniques researched in this study

Publication	Reference	IA Technique / Tool	Phase
Mining Version Histories to Guide Software Changes	Zimmermann, Weißgerber, Diehl & Zeller, 2005	A tool, ROSE as a plug-in to Eclipse IDE that analyses the evolutionary coupling based and guides developers about SLOs that should be changed depending on the historical changes to same dependency set.	EIS
Impact Analysis by Mining Software and Change Request	Canfora & Cerulo, 2005	Analyses the Change Request (CR) description in BugZilla and performs a free text analysis on history of	EIS

Repositories		CVS commits with same keywords using an algorithm. Eclipse plug-in *Jimpa* was developed in where the granularity is on file-level.	
Fine Grained Indexing of Software Repositories to Support Impact Analysis	Canfora & Cerulo, 2006	Analyses the Change Request description in BugZilla and performs a free text analysis on history of CVS commits with same keywords using an algorithm. The custom tool *Jimpa* was further developed. The granularity may be defined either on line or file level.	EIS
Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code	Kagdi, Gethers, Poshyvanyk & Collard, 2010	Proposes a technique where evolutionary and conceptual couplings are combined to determine EIS. The *sqminer* tool is mentioned for analysing evolutionary coupling but no complete automatic tool was developed.	EIS
Towards a More Efficient Static Software Change Impact Analysis Method	Jashki, Zafarani, Bagheri	Data-mining of VCS to determine clusters of files that tend to be changed together.	EIS
On the Precision and Accuracy of Impact Analysis Techniques	Hattori, Guerrero, Figueiredo, Brunet, Damásio	A coarse-grained static code analysis tool *Impala* was developed that analyses the dependencies between classes.	EIS
Empirical Software Change Impact Analysis using Singular Value Decomposition	Sherriff, Williams	A static impact analysis method utilizing singular value decomposition on association clusters of historical data	EIS
Impact analysis by means of unstructured knowledge in the context of bug	Torchiano	An approach to utilize Natural Language Processing techniques on version control log and code comments to guide impact	EIS

repositories		analysis	
An Eclectic Approach for Change Impact Analysis	Ceccarelli, Canfora, Cerulo, Penta	A study of using statistical analysis and Granger causality test to assess if historical time series data is useful when predicting future data of other time series	EIS

3.5 Summary of the previous research

Zimmermann et al. (2005) implemented a tool called *ROSE* that performs data-mining on version histories of VCS such as CVS. By analysing and recording the history of source code changes the tool may guide the developers during the implementation phase about related changes that were made earlier when changing the same files that are being changed now. The *ROSE* automatically determines the estimated impact set based on the historical actual impact sets that are data mined from the VCS. The tool may be useful for developers but not for the QA.

Canfora and Carulo (2005, 2006) introduced a tool that data mines the change request information from *BugZilla* and links the change requests to the actual source code changes that are revisioned in CVS. The tool may be used to find previously made similar changes to help the developers during the change impact estimation phase. While their research focuses on similar concepts and approaches the tool that they introduced may not be used for quality assurance purposes directly since the main usage is directed for determination of the EIS.

Kagdi et al. (2010) researched the subject of EIS determination through conceptual and evolutionary couplings. In their method the initial step is to determine the conceptual couplings for a first software entity that a change request is to be made. Then, the VCS is data-mined for previous changes made to the the entity that is being analysed to determine the evolutionary couplings. By combining the result sets from conceptual and evolutionary couplings the estimated impact set is determined. They utilized a tool called *sqminer* for data-mining of the software repositories. This tool does not seem to be available to the public domain.

Jashki et al. (2005) perform data-mining of CVS determine common couplings between files that are usually changed together. By gathering such sets the estimated impact set may be determined based on the historical data in the repository.

Hattori et al. (2008) developed a static code analysis tool called *Impala* which analyses the dependencies between classes. The empirical study that was performed used six dependency depth levels with three different case studies. The precision of the analysis decreases when the depth level increases (i.e. the size of the false poistive set increases). On the other hand the false-negative set decreases when the depth level increases.

Sherriff and Williams (2008, chapter 3) propose an interesting approach of collecting historical data of the changes and coupling the changes together into an association cluster of files. From the changed files a square matrix M is formed where the size of the matrix is the total amount of changed files. Each axis represents the files that are changed. The values are the amount of changes to each file and its related pairs. By calculating the singular value decomposition of the A the matrix U (and V) indicate the structure of the association clusters and the $\sum$ matrix represents the strength of the cluster, i.e. the amount of variability that each association cluster contributes to the matrix M. This approach is straightforward and requires minimal data mining when compared to the traditional static or dynamic impact analysis techniques as only the change clusters are analysed with SVD. The downside of the approach is that the change sets should follow a disciplined pattern where all of the related files are changed in single change set. In addition, if there is no historical data for a set of files the technique does not work.

All of the studied publications focus on the determination of estimated impact set and focus on aiding the developers before the change is made to the system. The data-mining and evolutionary couplings are in scope of the previous researches but they do not provide a solution for the actual impact set analysis. This leaves room for an IA technique and a tool that focuses on AIS, i.e. the changes that have already been made.

4 Problem Identification and Motivation

4.1 Software changes

Most of software development work is related in the software maintenance as introduced in Chapter 2. This implicates that the changes are being constantly performed on the existing system. Lindvall et al. (2013, 22-23) identify the system maintenance process to include the following sub-processes:

- Understanding of the change request, system and its structure
- Locating the primary and secondary parts of the system to be changed to implement the change request (i.e. defining the *estimated impact set*)
- Implementing the change request and tackling its ripple-effects
- Testing the newly incorporated changes and the system for regression (QA)

The case study by Lindvall and Sandahl (1998, 9-17), shows that even if the impact analysis is performed in a *formal* way the estimated impact sets tend to be greatly optimistic. The *actual impact set* tends to be a significantly larger than the initial estimate.

When impact analysis is performed in *informal* way the expertise and system knowledge of the developers determine the informal “estimated impact set”. The changes are done directly to the software life-cycle objects based on the change requests (CR).

By either *informal* or insufficiently complete *formal* process of impact analysis the end-result is that the software changes have a high chance of causing unexpected behaviour as ripple-effects in other affected parts of the system.

As the determining the estimated impact set is a very time-consuming (Lindvall et al. 2003, 49) and its results are usually non-complete (Lidevall et al. 1998, 9-17) a new approach is hereby proposed.

Instead of focusing on the estimated impact set the actual impact set should be analysed. From the actual impact set the information about actual software changes may be mined by techniques such as static code analysis. This allows

also the analysis of the ripple-effects by analysing the dependencies of the primary changed set of affected software life-cycle objects. The results of this analysis would provide a higher level of visibility of the software changes for the QA to concentrate the testing effort on.

4.2 The Enterprise software

The Enterprise software may be loosely defined as software that helps solving the enterprise-wide business needs of an enterprise (Enterprise Software, 2013). As the features and requirements for an enterprise software are usually unique to the particular field of business and solution there are certain characteristics that an enterprise software solutions have in common. Some of these characteristics include; however, are are not limited to the Reliability and Availability, Security, Scalability and Interoperability.

The Reliability and Availability characteristics define that the software should be readily available at all times and perform its tasks without a failure. The Security characteristics define that the software must handle and store the data securely and also prevent malicious use. The Scalability characteristics define that the software must scale when amount of users and possible integrations increase without critical hit to performance and other non-functional characteristics of the software (Wonders of the J2EE Architecture, 2013). Depending on the project these characteristics must also be tested.

4.3 The role of Quality Assurance

The quality assurance of the modern enterprise software is a challenge on multiple levels. The enterprise software itself consists of the features that solve the problems and fulfil the needs of a particular field of business. Enterprise software also often integrates with other systems which in turn raises the complexity. These features and integrations with other systems are defined by functional requirements. In addition, there are other characteristics that the system must have. These are defined through non-functional requirements.

The quality assurance process must include the testing of the fulfilment of the functional and non-functional requirements that are identified for the enterprise software. The level of testing that must be performed depends on

multiple factors. Some of these include the non-functional characteristics that are vital for the system at hand. To mention some as an example there are the performance, robustness, availability and security considerations (Non-Functional Requirement, 2013). These factors depend on the area of business that system is being developed for. For example, a system created for entertainment purposes has vastly different non-functional requirements than a system created for defence industry.

All these factors define the level of testing for a particular software that is necessary to perform in order to achieve high-quality enterprise software.

4.4 Developing Java Enterprise Edition Applications

The Java EE software is separated into different layers or modules with different responsibilities. In each layer a specific set of Java technologies may be used. The different specifications are defined by Java Specification Requests (JSR). The Java EE Platform Specification is an umbrella project that integrates the Java EE technologies together. For Java EE 7 this project is managed through JSR 342 (Java EE Platform Specification, 2013). The specification references other JSRs that specify the APIs that Java EE platform provides. There are several different implementations of those JSRs on which the actual Java EE software may be built on. An example set of essential Java EE technologies may be found in Table 2.

Table 2: Example of a set of essential JSR technologies for Java EE 7 applications

JSR	Description	Example providers
JSR-338 - Java Persistence 2.1	The Java Persistence API 2.1 (JPA) is the API for object-relational mapping and persistence functionality	Hibernate, EclipseLink, TopLink
JSR-345 - Enterprise JavaBeans 3.2	The Enterprise JavaBeans API (EJB) defines a component-based architecture for development and deployment of an	A supporting application server such as Oracle GlassFish 4, IBM WebSphere 8.5, Oracle WebLogic

	enterprise application. The EJB specification specifies how an application server provides responsibilities such as: • Transaction management • Integration with JPA • Concurrency control • Java Messaging Service • Asynchronous method invocation • Job scheduling • Naming and directory services, • Interprocess communication (RMI-IIOP and Web services), • Security: Java Cryptography Extension (JCE) and Java Authentication and Authorization Service (JAAS)	
JSR-346 – Contexts and Dependency Injection for Java 1.1	The Contexts and Dependency Injection API mainly provides: • Life cycle contexts for stateful objects for managing their life cycle • A dependency injection functionality where dependencies to beans may be injected either in development or	A supporting application server such as Oracle GlassFish 4, IBM WebSphere 8.5, Oracle WebLogic

	deployment time (Context and Dependency Injection for Java 1.1, 1)	
JSR-343 – Java Message Service API 2.0	Provides point-to-point and publish and subscribe style messaging facilities that enable integrations to other messaging systems (Java Message Service API 2.0, 12-13)	A supporting application server such as Oracle GlassFish 4, IBM WebSphere 8.5, Oracle WebLogic
JSR-344 - JavaServer Faces 2.2[1]	JavaServer Faces (JSF) is a UI framework for Java web applications. It provides facilities to create UI easily from a set of reusable UI components and also possibility to create custom UI components (JavaServer Faces 2.2 Final Release, Burns).	PrimeFaces, IceFaces, JBoss RichFaces
JSR-245 - JavaServer Pages 2.3[2]	JavaServer Pages (JSP) is a technology that provides capabilities to build dynamic web content such as HTML, XHTML and XML. A supporting application server such as Oracle GlassFish 4, IBM WebSphere 8.5, Oracle WebLogic	A supporting application server such as Oracle GlassFish 4, IBM WebSphere 8.5, Oracle WebLogic
JSR-341 – Expression Language 3.0	Expression Language is a language for presentation layer and may be used with JSF and JSP view technologies for easier access to data	A supporting application server such as Oracle GlassFish 4, IBM WebSphere 8.5, Oracle WebLogic
JSR-224 – Java API for XML-Based Web Services (JAX-WS) 2.2	JAX-WS specifies the XML based Web Services	Apache Metro
JSR-339 – Java API for RESTful Web Services	JAX-RS specifies the Representational State	Jersey, Apache CFX

1, 2 JavaServer Faces and JavaServer Pages are used only as an example. Several possibilities exist for creation of the front-end.

(JAX-RS) 2.0	Transfer based web services	
JSR-353 – Java API for JSON Processing 1.0	The JSON (JavaScript Object Notation) is a string-based data-interchange format. This API defines ability to manipulate and query JSON data.	JSON Processing (Reference impl.)

In Java EE application architecture the different responsibilities are defined in a layered model. These layers may include:

- The Domain Object Layer. Defines the data that is handled in the software through object-relational mapping (JPA)
- The Data Access Layer. Defines how the relational data is accessed. This is defined by Java Persistence API (JPA)
- The Service Layer. Defines the actual business logic of the software (EJB)
- The Web Services Layer. Exposes and provides the external services of the software. (JAX-WS, JAX-RS)
- The User Interface Layer. Defines the user interface for the software. (JSF, JSP, others)

By this modularity the software gains intrinsic values such as maintainability, scalability, unit testability and so on.

The main focus when developing Java EE applications is intended to be the business logic itself. When following the set of Java EE standards this may be achieved.

The Figure 2 illustrates an example of deployment of Java EE application with a separate Web Server.

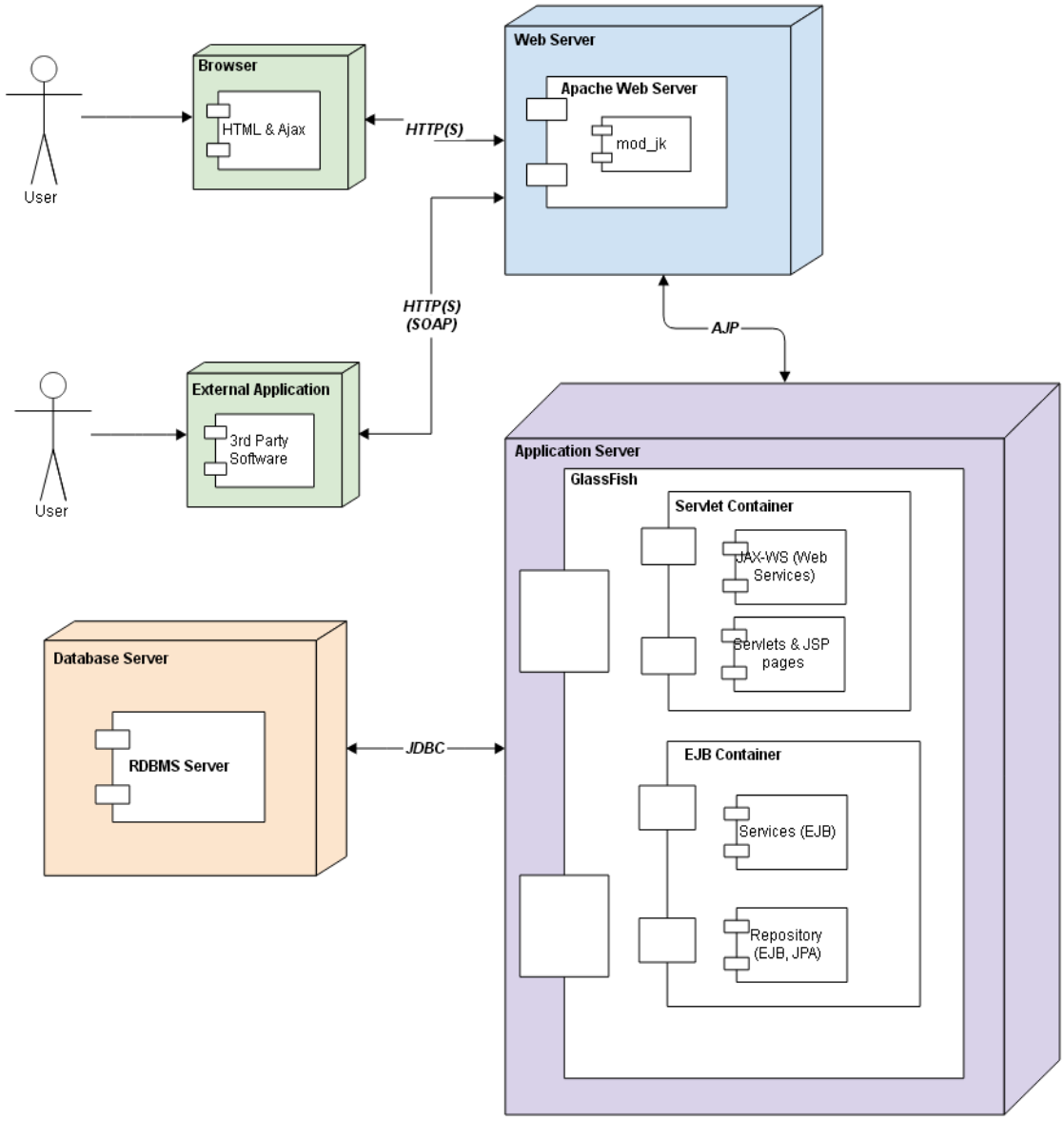


Figure 2: Example deployment diagram of a Java EE application

4.5 Testing of Software in Agile projects

In agile environment the testing is an integral part of the project. The quality of the deliverables is summarized by the success of all of the phases beginning from the defining of the user stories and ending up to the acceptance testing, preferably performed by the customer. In between there is the writing of unit-

tests, implementation of the features, integration tests and possibly exploratory testing and more. The single most important factor in succeeding in the quality assurance in agile projects is the collaboration between developers and other stakeholders in the project, particularly the customer. Depending of project size there may not necessarily be specific testers in the project but the developers act as testers as needed. The aim in Agile projects regarding testing is to accomplish most of the testing effort by writing automatic tests. (Myers et al., 2011, 178-179). Nevertheless the manual testing is usually required as there usually are too many different possibilities and variations for the automatic tests to capture (Whittaker, 2010, 14-15).

Two well-known testing strategies include the black box testing and the white box testing. In the black box testing strategy the tester has no knowledge of the internals of the system under test and the testing is based solely on inputs and outputs. In the white box testing the internals are known. Since exhaustive testing of inputs is not possible one needs to devise a testing plan and test cases that will include the most of the real inputs and outputs (Myers, Sandler & Badgett, 2011, 8-12).

The unit tests are an automated set of tests that test the functionality of each layer of the software. They test the class level implementations on a method-by-method basis. They do not test the integration of the layers and thus do not test the complete system. This phase is usually performed with white box testing methodologies. (Myers et al., 2011, 86)

The functional tests are either an automated or manual set of tests that test the integration of all the layers in the enterprise software with similar or exactly same test data that the actual system defines and uses. The purpose of these tests is to find out possible problems in co-operation between different layers. They test the actual features thorough the system top-down. Usually the user interface layer is not included in this test set. This phase is usually performed with black box testing methodologies; however white box may apply in certain cases. (Riley & Goucher, 2010, 200-201. Myers et al. 2011, 119)

It is also possible to test the functionality of the system at runtime by defining automated user interface tests which simulate the actions of the actual user. For example there is a a tool called *Selenium*. By using the tools the user

interfaces may be tested automatically with multiple browsers without human interaction. *T*he *automated UI tests* are a good way to test out the most important use-cases of the user. On the other hand they may be costly to implement and especially the upkeep is time-consuming as the software changes. Every time the user interface changes the automated UI tests must be updated as well. (Riley & Goucher, 2010, 200-202, 207)

The software may be tested by the approach called *exploratory testing*. In exploratory testing the software under test is not being tested by a set of pre-defined test cases but instead in a seemingly ad hoc manner. The tester may execute her intuition and expertise as seen fit based on the information provided by application. While this is mostly manual effort the automated tools are not ruled out in aid. In this technique the expertise of the tester is important to gain the most benefits. The test cases and results are defined on a feature-by-feature basis as the test are performed instead of writing them beforehand. The exploratory testing fits particularly well agile methodologies are applied in the project. (Whittaker, 2010, 16-17).

A study has proven that this method of testing does provide about an equal amount of defects found when compared to traditional test case based testing (Itkonen, Mäntylä, Lassenius 2007, 69).

4.6 The Dilemma

Since the need for the feature, or a change request arises from business-requirements of a customer there usually is a set deadline for the feature to be released. This means additional challenges for the project manager and development team including the quality assurance**.**

In the Figure 3 an Euler diagram of the Project Management Triangle defines three feats: "Good", "Fast" and "Cheap".

Figure 3: An euler diagram of project management triangle

In this model one may pick two traits that the project may have in opposition to the third option. For example if Good and Cheap are selected the end result will not be delivered Fast. If Fast and Cheap are selected the deliverables will not be Good.

In real-life projects experience has indicated that the Fast is usually one of the selected traits and the Cheap is the other (Dustin, Raska, Paul, 2007, 3). In this kind of project environment special considerations need to be taken into account to achieve satisfactory quality.

Since testing is a time-consuming process the amount and level of testing must be defined and the actual effort prioritized. Performing exhaustive testing in real-life projects is next to impossible to perform due to the amount of data variations. The aim of test manager is to define a finite set of test cases that finds the largest amount of errors. (Myers et. al. 2011, 9-10). If a test set is non-complete or the test engineer is unaware of the changed features that should be tested this will ultimately lead to regression, which means that old bugs might appear to the software again or that a feature that was previously working gets broken by a change to another feature.

4.6.1 Is there a solution?

To achieve a satisfactory quality the parts of the system affected by the change need to be identified on a feature-level and the limited testing resources must

be focused on these identified features. This may be a challenging task for both the developers and testers in large software projects.

To alleviate this problem the changes to the software must be tracked and analysed. For this purpose an impact analysis software is proposed. The purpose of this software is to analyse the changes to the enterprise software that were made by mining a set of commits from software repository on a feature-level. The software must also identify the components and services that the change affects. The set of changes must be tracked per version and must be given an impact score that is relative to other sets of changes. The impact score calculation is performed based on amount of files that are changed and ripple-effects analysed from dependencies of the changes. The higher the amount of files and dependencies, the higher the impact score (see chapter 6.3.5). From this list the highest impact, i.e. the highest priority changes can be identified. The changed components and their dependencies may also be investigated. This allows either the developers to write proper automated tests or the testers to properly test each change manually.

By utilizing this method the chance of regression is decreased and the understanding of the software change in a system increases iteratively.

5 Objectives for Solution

The primary objective is to provide a solution to the problem identified in Chapter 4. The list of requirements uses the MoSCoW-method for prioritization (Moscow Method, 2014).

5.1 Objectives for AIS analysis software

- The software should allow to configure multiple projects per instance
- The software must allow the configuring of software repository location per project. The targeted VCS system is Git as it is the most trendy VCS as of 2013 (Version Control Systems in 2013).
- The software changes must be automatically analysed for a project by mining the information from change-sets (i.e. commits) made into software repository

    - The software must allow configuration of the pattern that is used to mine the commit CR identifier. A regular expression is used.
    - The software must combine all the commits with same CR identifier into a single AIS
    - The analysis of affected SLOs must be performed on the feature level rather than commit level, i.e. all of the commits with same identifier must be combined in the scoring
- The features for a project must be determined based on automatic and user-defined mappings between SLOs and features
- The user could be able to follow the analysis process for a project in user interface and the user could have the possibility to stop it
- The user could be able to view the analysis status of a project
- The user must be able to filter the changed features of a project by following criteria:
    - Release Version: Showing all the changed features for a single version. Under each feature the list of commits that affected the feature are shown
    - Commit: Showing all the changed features by a commit identifier grouping all of the commits with same identifier under single set of changes
- The software could link the CR identifiers directly to the issue tracking system in use. In this case the software must support the JIRA issue tracking system. The URL to the issue tracking system must be configurable.

5.2 Data mining of the software repository information

The information about the software changes is contained with full history data in the software repository, e.g. Git. The changed SLOs are stored in a change-set, i.e. a commit. For one CR implementation there may be multiple commits.

By using a naming convention where the commit is marked with the CR identifier the commits may always be traced back to the original CRs when investigating repository history information. For a single CR all the commits with the same identifier prefix must be combined into a single AIS. After this the feature analysis may be performed for the AIS.

5.3 Feature definition approaches

To map the affected files to a specific feature some of approaches are hereby proposed:

- *The manual-mapping* approach as hereby defined where a user-entered pattern is mapped to a specific feature name. The patterns may be applied to the package names, file names and folders.
- The *convention-driven* approach as hereby defined the where package structure of the Java EE modules follow agreed coding conventions rigorously. An example of such conventions is the Oracle naming conventions (Naming Conventions 2014). A simple pattern matching algorithm should be able to be deduce the feature names from the package names and folder structure.
- A conjunction of the two previous approaches. In this case the software may initially attempt to find a set of features during the analysis by convention-driven approach. After this the user may manually correct and add the proper mappings of package, file and folder name patterns into features.

An example of a small part of an imaginary Java EE software is presented below to further elaborate the subject. The program called *Warehouse* is implemented for supply chain management for an imaginary company called *Warex*. The software handles the information of the contents and related meta-data of several warehouses globally.

In Tables 3 and 4 the convention-driven approach is presented by an example data set.

Table 3: Example of Java package and class names and automatically

Java Class Package Name	Java Class Name	Detected feature	Description
com.warex.warehouse.domain.receival	Receivement	Receival	A domain object representing information of a receival of goods
com.warex.warehouse.repository.receival	ReceivalRepository	Receival	A Peristence layer implementation of Receival domain objects
com.warex.warehouse.services.receival	ReceivalService	Receival	A service handling receival of items to warehouse(s)
com.warex.warehouse.web.receival	ReceivalCommand	Receival	A command object for user interface data transfer

Table 4: Example of folder location and automatically determined feature name

Directory name	Filename	Automatically detected feature name	Description
/WEB-INF/jsp/receival	ReceivalList.jsp	receival	Receival-list view for listing receivals of a one warehouse.
/WEB-INF/icons/receival	damaged-goods-warning.png	receival	Damaged goods Warning Icon

5.4 Change Analysis

The changes itself to the changed files may cause the undesired ripple-effects as explained in Chapter 2. To identify the changed code paths a couple of approaches are proposed:

- Static code analysis approach. In this approach the code baseline is analysed against the change-set. All affected methods are looked up for

references in other modules. The feature is determined for all the modules referring to the method in any level of call-stack and the feature is marked as 'Changed'.

- Dynamic code analysis approach. The software is analysed during run-time.

6 Design and Development

6.1 Overview and Goals

The tool that is implemented during writing of this thesis is called *J-Ace – Java Actual Impact Set Analyser*.

The initial approach of the development is to design the overall architecture and the design of the mandatory feature set. One of the secondary design goals is to use the standard Java EE libraries without usage of external libraries. This goal is set for the purpose of investigating how far the development of a Java EE application is possible without external dependencies such as Spring framework. The available frameworks and tools are studied and the libraries that are selected are based on following traits:

- Activity of the development of the library (the support for the future)
- Availability of documentation for the library (the velocity of development)
- References of the usage of a particular library (the public approval)

6.2 Architecture

The J-Ace is built on a selected set of core Java EE 7 technologies. The overall architecture consists of a multi-layered set of modules that each handle their specific responsibilities cohesively. The build life-cycle of the modules and their internal and external dependencies are managed by the Maven tool and Maven-specific Project Object Model (POM) files. Each module has its own set of dependencies that are needed for either in the compile-time, runtime or testing scope.

The project uses JPA 2.1 for persistence layer and EclipseLink is selected as

the persistence provider.

For user interface layer the Vaadin 7 is selected since it is possible to implement user interfaces with pure Java and thus fits particularly well for the purposes of the testing of the tool on itself for ripple effects.

The dependencies for each module are injected by inversion of control (IoC) framework provided by JSR-346.

The selected technologies and rationale are defined in greater detail in the Table 5.

The modules of the project are defined in the Table 6.

Table 5: The selected technologies for the tool

Technology	Purpose	Rationale
Java EE 7	The general technology platform for the software	The purpose of the software is to analyse Java EE software. In development of the software the certain design principles may be used to demonstrate the tool on top of its own source code.
Apache Maven	The build framework for managing the project configuration and controlling the build life-cycle	A well-known and solid tool for management of Java projects, their internal and external dependencies , build life-cycle, packaging etc.
Postgre SQL	Relational database	An open-source relational database
Oracle GlassFish 4	Application Server	The GlassFish is a free application server that fits for the needs of the tool.
EclipseLink	The object-relational model persistence framework	The experiences have shown that Hibernate has certain problems, such as the lazy-loading of entities when original entity manager is no longer available (Hibernate Lazy Loading 2014). The EclipseLink was chosen for easier development. The persistence framework may be easily changed to any other JPA2.1 framework at a later time if a need

		arises.
Vaadin 7	User interface layer	With Vaadin 7 the developer may implement the UI with Java code. For the purpose of the demonstration of the tool itself it is an ideal choice for detection of ripple-effects in Java code.
JGit	Java API for GIT	The JGit was chosen since it is also the basis for Eclipse EGit implementation and has active development community. The problem is that the documentation is very lacking.
ASTParser	A Java language parser for creating abstract syntax trees (ASTs).	The ASTParser is used in Eclipse IDE for the purpose of analysing Java code. It has good documentation and has an active development community.
Apache Commons	General utility libraries for Java development	This is a tried and true library that has many of the common problems already solved that are faced in Java development. It has good documentation and has an active development community.

Table 6: Modules of J-Ace project

Module	Responsibilities	Main technologies
build	The Maven Parent project defining the common dependencies of the project and basic build life-cycle	• Maven
common	Common utilities used in J-Ace development	
dao	Data access layer. Retrieving and storing of entities from relational database.	• JPA 2.1 • EJB 3.1
domain	Defines the domain-specific entities of the J-Ace tool	• JPA 2.1
ear	Configures the EAR packaging of the project for deployment to	

	application server	
services	Contains EJBs for business logic of the J-Ace tool	• EJB 3.1 • JGit • ASTParser
web	Contains the user interface for J-Ace tool	• Vaadin 7

6.3 Implementation

This chapter focuses on highlighting the main design and implementation related choices in J-Ace project. The scope is restricted in impact analysis related functionality only.

6.3.1 User Interface

The user interface is implemented using *Vaadin 7*. The *Vaadin* is an open-source Web application framework in which most of the user interface logic is executed in server-side. To provide a rich user experience the framework uses *Ajax* and *Google Web Toolkit* for the client-side. The user interface may be implemented almost purely in *Java*.

The *Vaadin* supports Server-push that enables the client-side to automatically show the changes that are made in server side without additional need for custom coding.

The J-Ace UI is divided in three main views that are defined in Table 7.

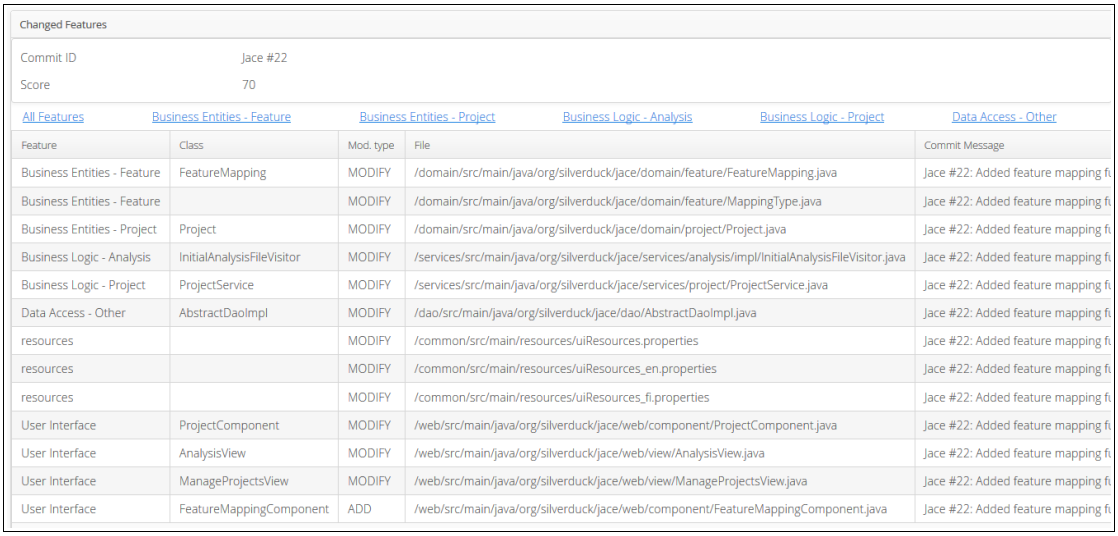
Table 7: J-Ace User Interface Views

View	Purpose
Analysis View	The Analysis view shows the performed analyses on configured Projects and Analysis settings. The user may view the changed features of a Project by Release version or by a combined set of Commits. The set is combined based on Commit Identifier. For example if a developer adds three commits into software repository with a commit message containing the identifier “EXAMPLE-3” as the prefix these three commits are regarded as one 'Commit' set in J-Ace for detection of changed features.
Manage Analysis Settings view	The Manage Analysis Settings view provides the user possibility to configure different Analysis for a configured

	Project. The settings define the following traits of an Analysis to be performed: • Branch – The Branch in VCS to analyse • Granularity – File- or method level granularity • Automatic Feature Mapping – To control does the J-Ace attempt to identify feature names automatically • Analysis Enabled – To control if Analysis is performed periodically for a Analysis setting
Manage Projects View	The Manage Projects View allows the User to configure Projects. A Project contains following basic set of properties needed for Analysis: • Name – The Name of the Project shown in J-Ace • Repository Type – The repository type. The J-Ace tool supports Git as the repository type • Remote Repository URL – The URL for the Repository to be used for Cloning • User Name – Optional user name if the Software Repository requires authentication • Password – Optional password if the Software Repository requires authentication • Commit ID Pattern – A Reguar Expression pattern used to identify a Commit Identifier. For example J-Ace uses Commit Identifiers that match the Issue identifiers in issue tracking system. The identifiers used are prefixed with “Jace” and suffixed with a running issue number *n*, for example “Jace 3”. A regular expression pattern for such Commit Identifier is “Jace #(\d)+”. In addition, the Release-version related configuration options are defined: • Version File Type – Either a *Properties* -file or *XML-file*. • Relative Path to Version File – Defines the file in Software Repository that contains Release-information for the project • Version Pattern – Defines the *property* or *XPath* to version information. This depends on the setting *Version File Type*. For exaple for a standard Maven project this would be “/project/version”. The Feature Mappings may be defined on the project level. Each mapping constists of: • Type – Java Package name, file name or directory

	name • Pattern – A regular expression pattern • Feature Name – The feature name that is used when pattern matches

The figure 4 illustrates the changed features screen under the analysis view where the features may be examined that were changed by a set of commits identified with the same identifier. By selecting a feature the table is filtered to show only the files that were changed from a feature. By selecting a Java-source file in the list the user may examine the dependency tree of the selected file as illustrated in figure 12 on page 50.


Changed Features

Commit ID Jace #22

Score 70

All Features | Business Entities - Feature | Business Entities - Project | Business Logic - Analysis | Business Logic - Project | Data Access - Other

Feature	Class	Mod. type	File	Commit Message
Business Entities - Feature	FeatureMapping	MODIFY	/domain/src/main/java/org/silverduck/jace/domain/feature/FeatureMapping.java	Jace #22: Added feature mapping fi
Business Entities - Feature		MODIFY	/domain/src/main/java/org/silverduck/jace/domain/feature/MappingType.java	Jace #22: Added feature mapping fi
Business Entities - Project	Project	MODIFY	/domain/src/main/java/org/silverduck/jace/domain/project/Project.java	Jace #22: Added feature mapping fi
Business Logic - Analysis	InitialAnalysisFileVisitor	MODIFY	/services/src/main/java/org/silverduck/jace/services/analysis/impl/InitialAnalysisFileVisitor.java	Jace #22: Added feature mapping fi
Business Logic - Project	ProjectService	MODIFY	/services/src/main/java/org/silverduck/jace/services/project/ProjectService.java	Jace #22: Added feature mapping fi
Data Access - Other	AbstractDaoImpl	MODIFY	/dao/src/main/java/org/silverduck/jace/dao/AbstractDaoImpl.java	Jace #22: Added feature mapping fi
resources		MODIFY	/common/src/main/resources/uiResources.properties	Jace #22: Added feature mapping fi
resources		MODIFY	/common/src/main/resources/uiResources_en.properties	Jace #22: Added feature mapping fi
resources		MODIFY	/common/src/main/resources/uiResources_fi.properties	Jace #22: Added feature mapping fi
User Interface	ProjectComponent	MODIFY	/web/src/main/java/org/silverduck/jace/web/component/ProjectComponent.java	Jace #22: Added feature mapping fi
User Interface	AnalysisView	MODIFY	/web/src/main/java/org/silverduck/jace/web/view/AnalysisView.java	Jace #22: Added feature mapping fi
User Interface	ManageProjectsView	MODIFY	/web/src/main/java/org/silverduck/jace/web/view/ManageProjectsView.java	Jace #22: Added feature mapping fi
User Interface	FeatureMappingComponent	ADD	/web/src/main/java/org/silverduck/jace/web/component/FeatureMappingComponent.java	Jace #22: Added feature mapping fi

Figure 4: A screenshot from J-Ace illustrating the changed features of a change set with ID Jace #22

6.3.2 Software Repository Functionality

The J-Ace tool supports Git Version Control System (VCS). The interface is defined in *GitService* class and logic is implemented in *GitServiceImpl* class. The *GitService* implements the *Plugin*-interface and thus provides methods for cloning, pulling, changing branches and listing branches in a software repository. The implementation uses open-source *JGit* library for git operations.

The local copies of configured software repositories are stored in server side in a directory that is configured in J-Ace

"*common/src/main/resources/jace.properties"*-file under the property *workingDir*.

The Git Service is responsible for parsing the *diffs* as specified in *Gnu DiffUtils Unified Format*. The data is stored temporarily into J-Ace database for later analysis to determine the features that have been changed.

6.3.3 Java Code Parser

The J-Ace analyses files specified with "java"-extension by using a custom implementation that utilizes the *Eclipse JDT*-library an more specifically the *ASTParser*. The *imports*, *field*- and *variable* definitions, *methods* and *method*-parameters are parsed and stored to J-Ace database for later analysis.

6.3.4 Analysis Service

The Analysis Service contains logic for analysing the files of a project after the initial *cloning* of the Software Repository and always after the changes have been *pulled* from VCS by the *Git Service*.

The Analysis Service has two kinds of analysis logic. The *initial analysis* is performed when an *Analysis Setting* is defined and saved. The initial analysis analyses all files in the configured branch. A data model is created and stored into J-Ace database that contains the Java-classes and other project files. The initial dependencies between classes are analysed at this phase. If *automatic feature mapping* has been defined in the analysis setting also the features are created at this phase. The logic for naming the features depends on the type of the file and its location. The feature name for the Java source files is determined by the package name of the class. This is the *convention-driven* approach as defined in Chapter 5.3.

The level of dependency analysis depends on the *Granularity* setting defined under the Analysis Setting. For file-level Granularity only file-level dependencies are analysed. The dependencies are determined from the import definitions in a class file. Only the internal project scope dependencies are analysed. External dependencies to other libraries are omitted.

For method-level Granularity the methods and the contained method-

invocations are analysed and dependency information that is analysed is more fine-grained and reduces the chances of false positives.

The Analysis is implemented in two phased approach. In first pass the imports are stored temporarily into J-Ace database. After all of the files are analysed in the first pass the second pass analyses all of the dependencies between them.

6.3.5 Scoring of Changed Features

The scoring of changed features that are detected in analysis phase is calculated by the following equation:

$$S = 2a + b + \sum_{i=2}^{n} \frac{c_i}{i}$$

In which the *S* indicates the score, the *a* indicates the amount of directly changed files, *b* indicates the amount of files that depend on the directly changed files. The c_i indicates the amount of files that depend on the files at *i-1* of each sub-sequent level where the *i* indicates the depth.

Given an example for calculation of score on file-level granularity in a scenario where the classes *B and* C depend on class *A*. The classes *D* and *E* depend on the class *B*. The class *F* depends on the class *C*.

The class *A* has changed hence the *a=1* and subtotal is 2. For the classes *B* and *C* the score is added by one for each dependency to class *A*, i.e. *b = 2*. Thus, the subtotal is 4. The score for the classes that depend on class *B* is calculated. The classes *D* and *E* add the score of 0.5 for each, i.e. *i=2* and c_i *= 2* making the sub-total of 5. The classes that depend on class *C* are calculated. The class *F* adds the score by 0.5, i.e. *i=2*, c_i*=1,* making the final score of *S=5.5*. In the user interface the *S* is rounded up and shown as integer.

6.4 Sequence Diagrams

6.4.1 Creation of the Project

The figure 5 illustrates the process of adding a new project to the system. The *ProjectService* calls *GitService* to clone the repository based on user-entered project configuration. After cloning the branches are queried from GitService

and updated to the project. The project is then persisted into database using *ProjectDao*. In exceptional situations the changes are rolled back and user is notified with an error notification and an error message describing the problem.

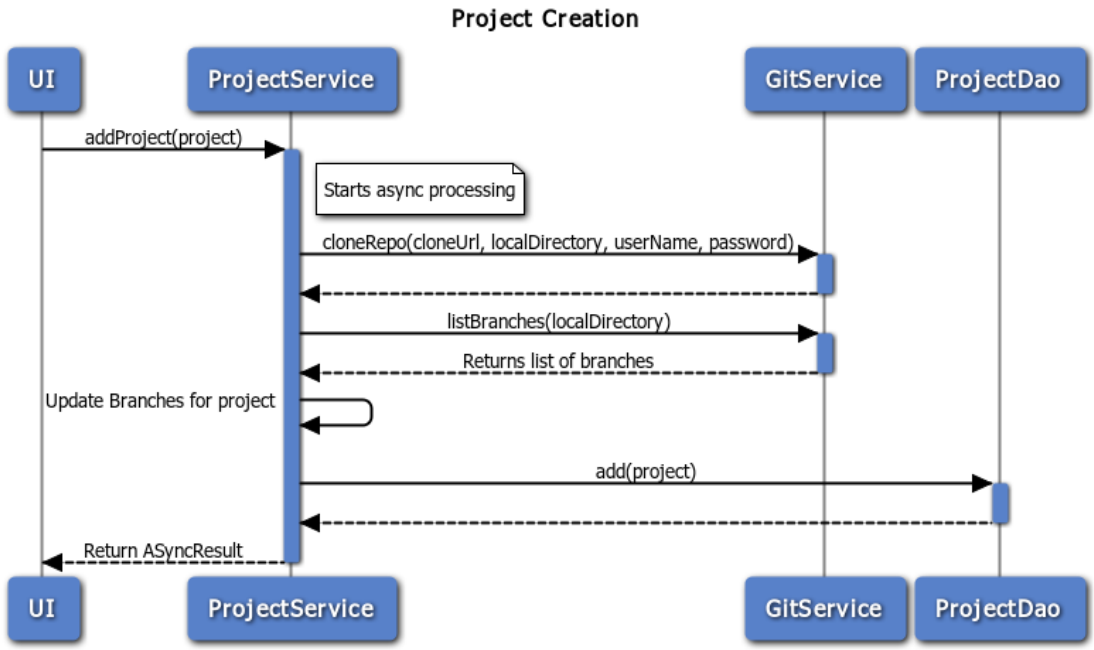


Figure 5: Sequence diagram of the project creation

6.4.2 Initial Analysis

The figure 6 illustrates the process of adding a new analysis setting for a project and the initial analysis process. The most important class is the *InitialAnalysisFileVisitor* that uses *ASTParser*-library to parse the Java code files and generate appropriate software life-cycle objects and related entities from them.

The feature mappings are checked against each file, directory and Java-class that are analysed. If a match is found the feature name is then resolved from the user-entered data. If a mapping does not match and automatic feature mapping is enabled the J-Ace gives a generated name for the changed feature.

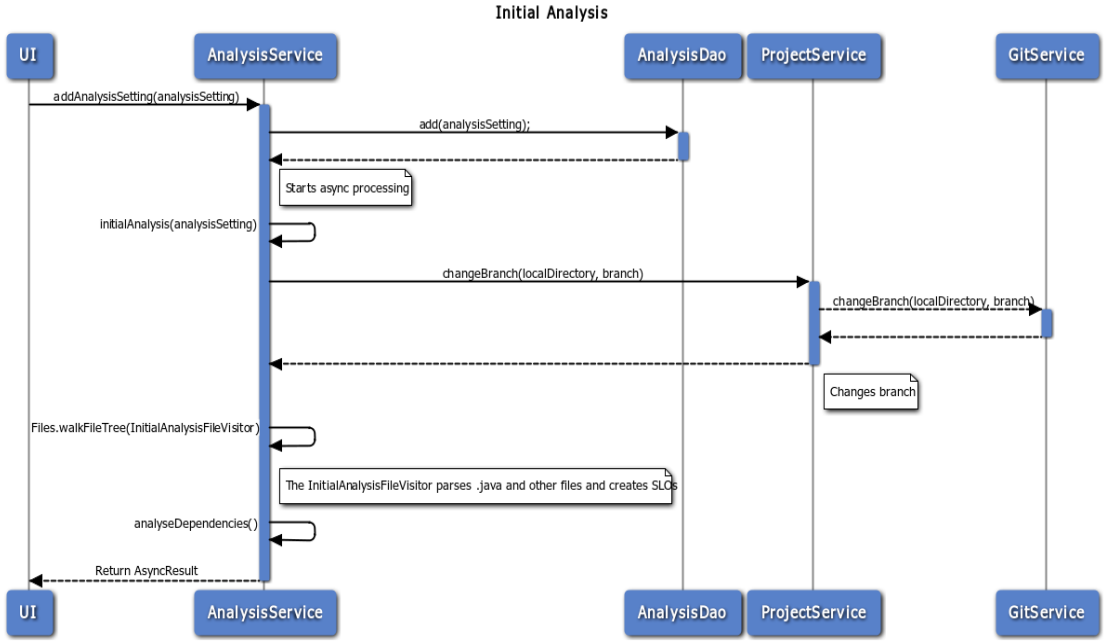


Figure 6: Sequence diagram of the initial analysis

6.4.3 Sub-sequent Analyses

The sub-sequent analyses are performed periodically by a timer. The *AnalysisService* calls the *ProjectService* to pull the changes from the remote repository. The *GitService* is called by the *ProjectService* for projects that are defined to use GIT project type. The *GitService* performs the pull-operation and then analyses the *GNU Unified Diffs* that define the changes that have been made to the files (SLOs) in each change set. It pre-processes them and returns the diffs as *Diff* objects to the *AnalysisService*. The *AnalysisService* then analyses the *Diff* objects and determines the features that have been changed. These are stored into *ChangedFeature* table in database.

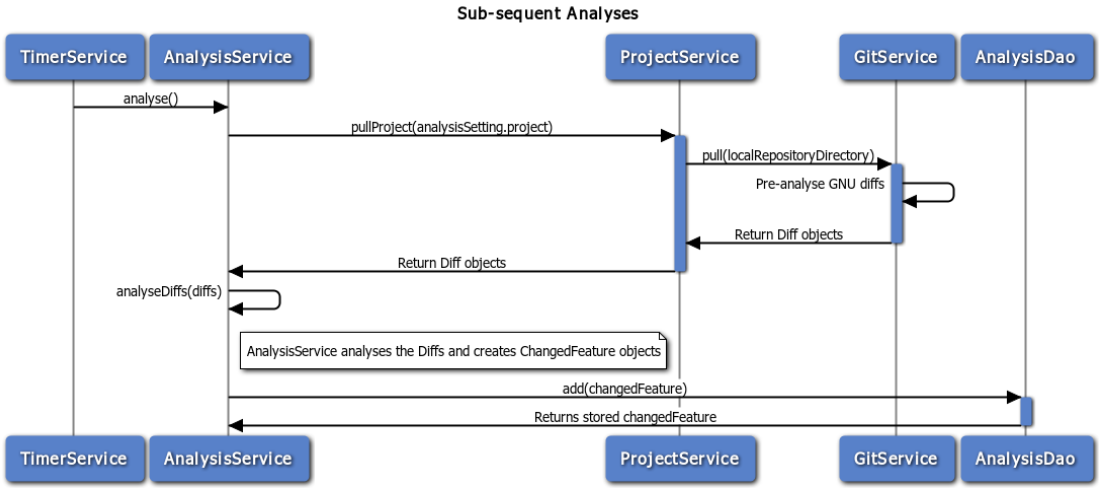


Figure 7: Sequence diagram of the sub-sequent analyses

7 Case Study: J-Ace

The first case study analyses the functionality of the tool by a known and controlled set of test data. The J-Ace tool is used to analyse changes made into itself. The modules that the project contains are defined in Table 8.

The case study consists of following phases:

1. Add the project into J-Ace
2. Add a new Analysis Setting that defines the Analysis to be performed on the project
3. Create change requests as issues into GitHub project issue management system. The nature of the change requests are defined in Table 8.
4. Implement the changes to the software
5. Commit the changes into GitHub
6. Perform Analysis in J-Ace
7. Analyse the Changed Features and Scoring

Table 8: The nature of Change Requests for demonstration

Issue Style	Example	SLOs	Dependencies	Expected Score
An issue that affects few files that have low amount dependencies	User interface change affecting one view	Few	Low	Low
An issue that affects several files that have low amount of dependencies	User interface change affecting multiple views	Several	Low	Moderate
An issue that affects few files that have high amount of dependencies	A change to an utility class that is used thorough the system	Few	High	Moderate
An issue that affects several files that have high amount of dependencies	A change to an entity object that affects the data access layer, the business logic in service layer and the user interface	Several	High	High

7.1 Project definition

The project may be cloned from a public git repository hosted on the *GitHub*. The basic project data includes the URL of the remote repository is entered in the project definition screen shown in Figure 8. Another notable field is the "Commit ID Pattern" that defines how the commits are identified and later grouped as defined in chapter 5.1. The commit pattern is defined as "Jace #(...)#". This enables the software to parse the commit identifiers from the commit messages and group all of the commits with same identifier into same changed feature set.

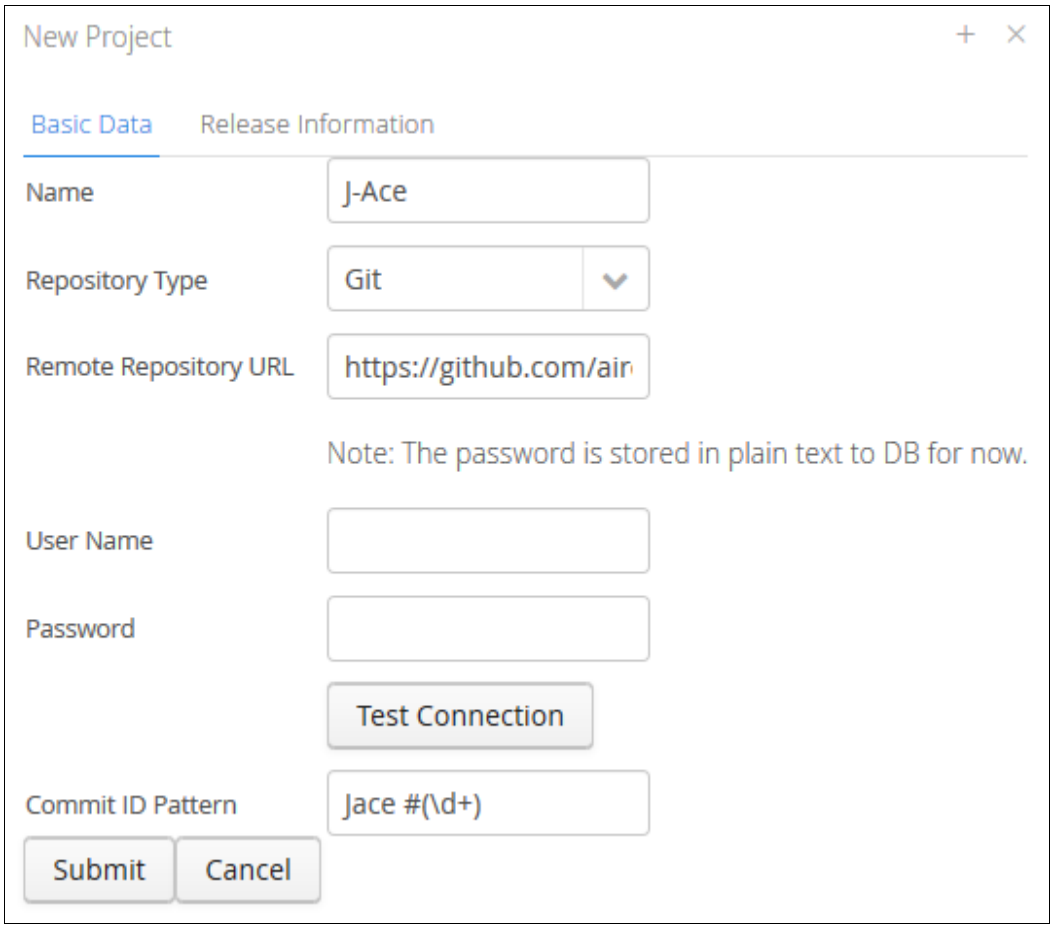


Figure 8: Basic project data definition screen

The J-Ace project uses *Maven* and thus contains the *Project Object Model*-file. The file contains the version number of the software. In the "Release Information" page the user may enter the type of the version file, the relative path to the version file and the *XPath*-pattern used to extract the version information from the file as illustrated in figure 9.

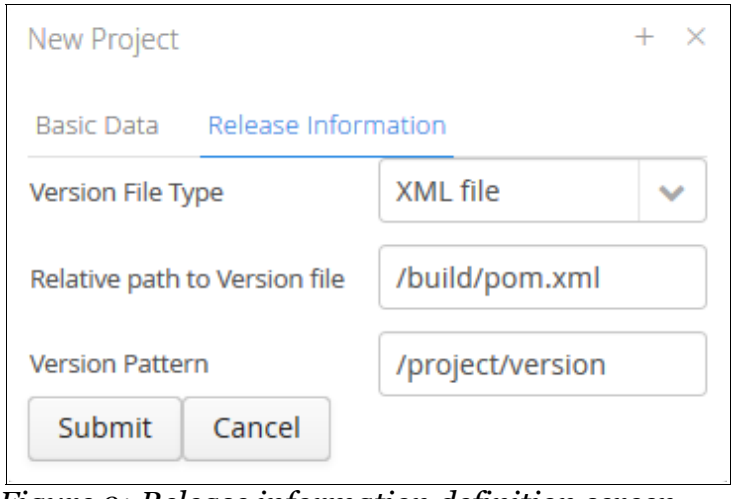


Figure 9: Release information definition screen

To define the features that J-Ace contains the feature names are mapped to directory and Java package names that are specific to the J-Ace project. Partial feature mappings are illustrated in the Figure 10.

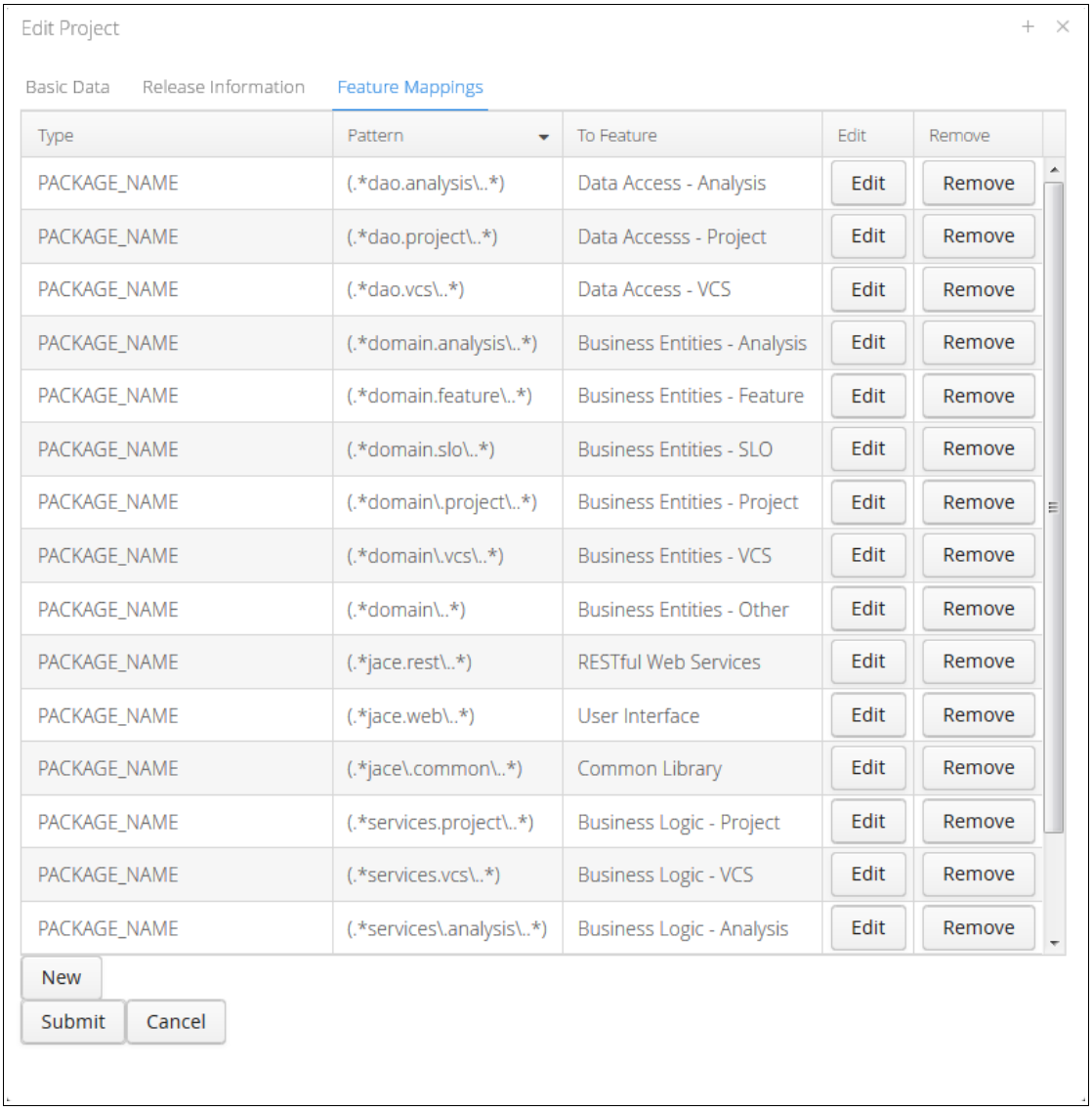


Figure 10: The feature mappings of the J-Ace project

7.2 Analysis Settings Definition

Once the project has been defined the user may define the analysis configuration for the project. There may be several configurations for a single project, for example for the purpose of analysing different branches for the same project.

The creation of a new analysis configuration is illustrated in the figure 11. The selected branch for the analysis is set to "*refs/remotes/origin/master*". The granularity is set to File-level and the *Automatic Feature Mapping* is enabled.

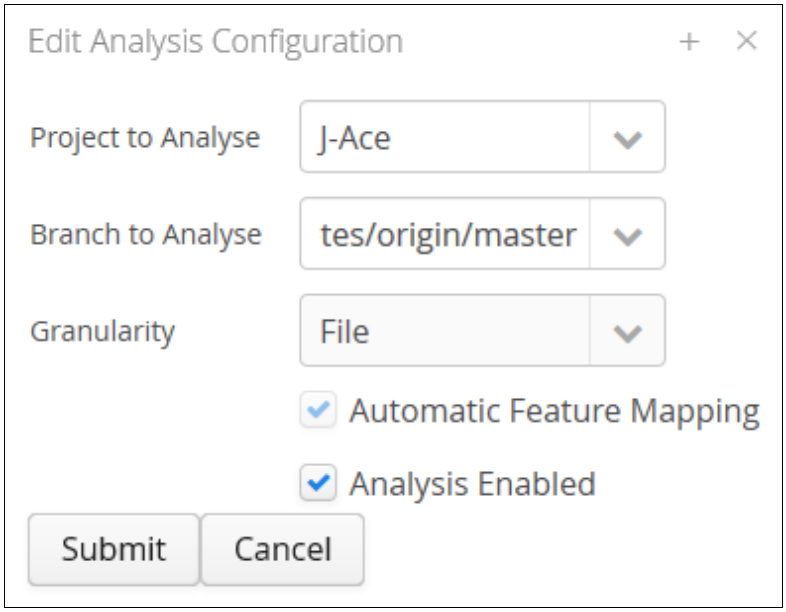


Figure 11: The Analysis Configuration screen

7.3 Issues

7.3.1 Few changes with low amount of dependencies

The J-Ace issue number 20 is defined as follows:

- Change the version number of J-Ace to version 0.4 and ensure that it is displayed in the user interface correctly

This isolates the changes to one file. The version number is defined in the *pom.xml*. The file path is shown in table 9.

Table 9: List of changed files

File	Type
build/pom.xml	Project Object Model-file

7.3.2 Several changes with low amount of dependencies

The J-Ace issue number 12 is summarized as “Remove UI Labels above Tables” and is defined as follows (excerpt from Issue Management system):

- Remove the UI labels above the tables. The labels are redundant, for example the 'Analyses', and 'Releases. Remove 'em all
- In Analysis View Change 'Details' group caption to 'Changed Features'

This isolates the changes to user interface code and localization messages but the changes are implemented to several files.

The implementation phase changes the files listed in Table 10.

Table 10 - List of changed files

File	Type	Is dependency of
AnalysisView.java	Java source code	ManagementToolbar.java
ManageAnalysisSettingsView.java	Java source code	ManagementToolbar.java
ManageProjectsView.java	Java source code	ManagementToolbar.java
UIMessages.properties	Resource file	-
UIMessages_en.properties	Resource file	-
UIMessages_fi.properties	Resource file	-

The score that is calculated should be the amount of directly changed files multiplied by two. In addition, the *ManagementToolbar.java* depends on each of the Views, so the score should be added by one for each dependency, hence the score should be calculated to 15.

7.3.3 Few changes with high amount of dependencies

The J-Ace issue number 21 is summarized as “Add Javadoc to common-project” and defined as follows (excerpt):

Add missing Javadoc to all of the classes in the project

The common-module is used thorough the system. Since the Granularity is set

to file level the J-Ace cannot detect that the changes are related only to *Javadocs* of the classes, i.e. it detects the whole files as changed. This suites the demonstration purposes well. It also indicates the inaccuracy of static code analysis when granularity coarse.

7.3.4 Several changes with high amount of dependencies

The J-Ace issue number 25 is summarized as “Clean-up and refactor J-Ace codebase” and defined as follows (excerpt):

> Make a pass through the whole code and refactor, clean-up and document as necessary

Most of the files in the project are altered in this change request.

7.4 Gathering the results

The table 11 illustrates the results from the four of the issues that were defined in, implemented and analysed. The levels indicate the dependency depth of the SLO in relation to the other classes, i.e. the other classes that depend on the SLO. The figure 12 illustrates an example of a dependency graph captured from the *J-Ace* user interface. It shows the classes that depend on the class *AppResources.java* at different levels. This file is from the *Jace #21* change set.

The results indicate that the commit sets that have higher amount of dependencies have similar scores to commit with larger amount of direct dependencies. The Jace #12 has six direct changes and and Jace #21 has only five direct changes. Due to the amount of dependencies the score of the Jace #21 is close with Jace #12.

The *Jace #25* commit has a score of 158. This is due to high amount of direct changes as well as dependencies on multiple levels. The score is the highest of the four issue types as expected.

Table 11 - Results of analysis for J-Ace

Commit ID (AIS)	Changed files	1st level	2nd level	3rd level	4th level	Score
Jace #20	1	0	0	0	0	2
Jace #12	6	3	3	0	0	17
Jace #21	5	5	1	0	0	16
Jace #25	52	25	12	15	12	158[1]

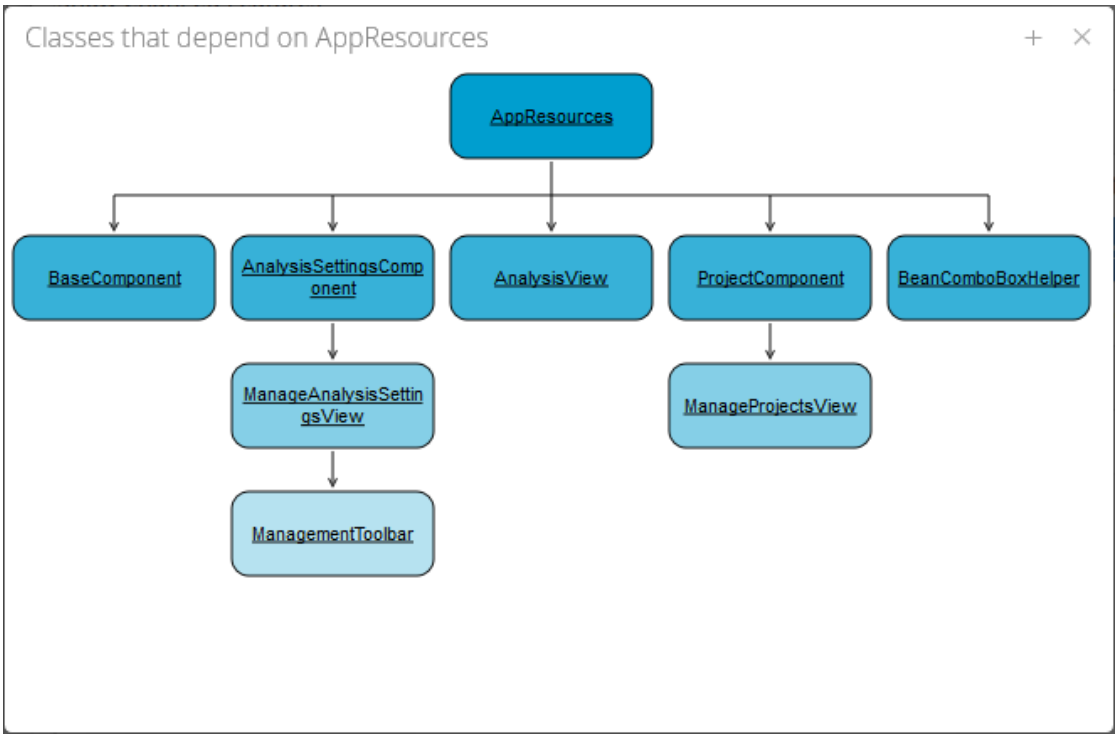


Figure 12: J-Ace screenshot of the classes that depend on the AppResources class

1 The Jace #25 contains also dependencies on levels 5..8 as follows. c_5=28, c_6=29, c_7=21, c_8=4

8 Case Study: Alfresco Community Edition

The *Alfresco* is an open-source *Enterprise Content Management* system. It was selected as the subject for case study for the following reasons:

- It is an actively developed enterprise application with a wide user base
- The existence of user interface layer is particularly important for the case study
- It is open-source
- It has an open issue management system

The case study consists of following phases

1. Add the project into J-Ace
2. Set git history to commit 26th of February 2014 to the commit before the first change that we want to analyse (ACE-716). This is performed with *git* command line tool.
3. Define the analysis setting
4. Perform initial analysis. This will analyses the 26th of February 2014 as the baseline to which the changes are compared.
5. Run the Analysis. This will detect any changes between 26th of February 2014 and current date.
6. Compare the selected issue management system issues against the results from J-Ace

The general approach is to assess how the issue itself describes the change request and then compare the assessment against the results from J-Ace.

8.1 Project definition

The project is defined with the settings described in the Table 12.

Table 12 - Alfresco project definition

Name	Alfresco
Repository type	GIT
Repository URL	https://github.com/Alfresco/community-edition.git
Commit ID pattern	ACE-(\d+)
Version file type	XML
Relative path to version file	/pom.xml
Version pattern	/project/version
Manual version mappings	Not defined

8.2 Analysis Settings definition

The table describes the settings that are used for analysis.

Table 13 - Alfresco analysis settings definition

Branch to Analyse	refs/remotes/origin/master
Granularity	File
Automatic feature mapping	Enabled

8.3 Issues

8.3.1 Issue that describes a bug in production release

The *ACE-3373* summarizes the issue as "*Approx transaction indexing time remaining: being incorrect*" with following description (excerpt):

> Approx transaction indexing time remaining value displays incorrect time in http://st1:8080/solr4/#/alfresco and http://sn1:8080/solr4/#/alfresco (e.g. only 10 minutes since reindex process is going on more than 3 hours).
> Please, see attachment with screenshot.

The issue also contains a screen shot of the UI that shows invalid time.

8.3.2 Issue that should change only the user interface layer

The *ACE-716* is summarized as “No way to sort results after performing Advanced Search” and described as follows:

> STR:
> 1. Go to Advanced Search page;
> 2. Perform a search;
> 3. Try to sort the results on Search page
>
> **Act. result:** no option to sort the output;
> Though 'Sort by:' option is present when performing simple search.
>
> Worked Ok on Alfresco Ent. 4.3 b135

The changes should affect only the user interface.

8.3.3 Issue that should change only single feature

The ACE-1857 is summarized as “Infinite scroll only gets one additional set of data” and described as follows:

The faceted search page is supposed to use an infinite scroll approach in order to retrieve additional pages of data. However, it currently only retrieves one additional page.

> To reproduce:
>
> 1. Make sure there are enough documents named so that more than 50 will appear for a given search term
> 2. Enter that search term into the search box and hit enter (the faceted search page should load showing the first 25 results
> 3. Scroll to the bottom of the page (the next 25 results should load)
> 4. Scroll to the new bottom of the page (no more data is loaded)

The comments in the Alfresco Ticket management system suggest that the issue was hard to reproduce and the approach for fix is not clear. This is a good candidate for a issue that could affect other parts of the system than intended.

8.4 Gathering the results

The results of J-Ace analysis are contained in the Table 14.

Table 14 - J-Ace results for ACE-1614

Issue	Direct changes	1st level	Score
ACE-3373	7	3	17
ACE-716	1	0	2
ACE-1857	6	0	12

8.4.1 ACE-3373

The ACE-3373 implementation fixes the estimated remaining indexing time calculation that is shown in the user interface. The reported of the issue is the Alfresco QA team and the severity has been marked as critical. The QA team performed the testing after the implementation verifying that the issue was resolved.

The results of J-Ace analysis indicate that the developer has done two commits for the issue with total of seven direct changes. The commits and their changes are listed in Table 15. (The spelling errors in the commit messages are not corrected).

Table 15 - List of commits and files of ACE-3373

Commit Message	File	Type
Fix for ACE-3373 Approx transaction indexing time remaining: being incorrect. - Fix incorrect tracking of nodes/transaction	MetadataTracker.java	Modify
Fix for Bug ACE-3373 Approx transaction indexing time remaining	AlfrescoCoreAdminHandler.java	Modify

being incorrect: - node and acl estimates now use elapsed time - added estimated time to finish for content index based on elapsed time - time now include query, tracking loop, index commit and index warming amortized per unit of work (node, acl or content)		
Fix for Bug ACE-3373 Approx transaction indexing time remaining being incorrect...	TrackerStats.java	Modify
Fix for Bug ACE-3373 Approx transaction indexing time remaining being incorrect...	AclTracker.java	Modify
Fix for Bug ACE-3373 Approx transaction indexing time remaining being incorrect...	ContentTracker.java	Modify
Fix for Bug ACE-3373 Approx transaction indexing time remaining being incorrect...	MetadataTracker.java	Modify
Fix for Bug ACE-3373 Approx transaction indexing time remaining being incorrect...	ContentTrackerTest.java	Modify

The features for the that J-Ace identifies automatically are called '*solr*' and '*tracker'*. These are very close matches to the actual features that have been changed, even though the feature names are on a rather fine-grained level.

The changes were further analysed using the *git diff* tool to identify the file-level changes. The added code uses elapsed time for the estimation of the remaining time. The code, comments and variables contain spelling errors which raise questions about the quality of the changes. These are invisible to the quality assurance team unless the team has knowledge and time to use the *git diff* tool.

While the changes seem to be constrained to the features that relate to the

estimation there might be unforeseen side-effects to the indexing feature itself if the estimation code malfunctions. It should be noted that the external *git* tool had to be used to gain this information.

8.4.2 ACE-716

The automatic feature mapping of J-Ace determines that feature called *'search'* is changed. No other features have been changed. The changes are constrained to a single file called "*search.js*". As this file is a JavaScript file the J-Ace cannot currently parse any dependency nor usage information from it.

The J-Ace succeeds in confirming that the issue had changed only a file that relates to the user interface. No other changes have been done, and only the search UI should be tested.

A manual inspection performed on the file with *git diff* -command using the previous commit hash and the commit hash of the ACE-716 issue show that the developer had added only four if-clauses against a variable called '*ToggleLink*'. This further confirms that the change is constrained in the user interface level and should not cause side-effects.

8.4.3 ACE-1857

The J-Ace analyses that there are three commits related to fixing this issue and total of 6 changes. The automatically detected feature names are '*documentlibarary'* and *'layouts'*. The commits and files are listed in Table 16 in which the Files listed are relative to directory '*/project/slingshot/source/web/js/alfresco/documentlibrary/*'.

Table 16 - List of commits and files of ACE-1857

Commit Message	File	Type
ACE-1857 : Ensure that infinite scroll on faceted search gets more than just one more page of data.	AlfDocumentList.js	Modify
ACE-1857 : Ensure that	views/layouts/_MultiItemRendererMixin.js	Modify

infinite scroll on faceted search gets more than just one more page of data.		
ACE-1857: Exploratory commit to see if it fixes DP infinite scroll issue	AlfSearchList.js	Modify
ACE-1857: Prevent infinite scroll triggering multiple times for the same page load	AlfDocumentList.js	Modify
ACE-1857: Prevent infinite scroll triggering multiple times for the same page load	AlfSearchList.js	Modify
ACE-1857: Prevent infinite scroll triggering multiple times for the same page load	_AlfDocumentListTopicMixin.js	Modify

The J-Ace cannot parse the JavaScript-files to detect possible dependencies.

The user may confirm from the J-Ace user interface that the changes have been performed solely on the user interface code and possible side-effects should be constrained there.

9 Evaluation

9.1 The J-Ace Features

The J-Ace tool accomplishes the primary objectives that were set (see chapter 5.1). The tool may be used to evaluate the software changes of a Java EE software project and more importantly to evaluate the most important changes made to the system per release. To identify the most important issues the user may inspect the list of commits per release and look for the impact score adjacent to each commit identifier or message. If the commit identifier could not be parsed (or does not exist) the commit message is shown instead. The commits are sorted by impact score in descending order, i.e. the most important change is listed first. The tool exposes the potentially most hazardous commits by the impact score and guides the quality assurance to focus on testing these issues for possible side effects.

In addition, the user may filter the results by the following criteria:

- By a release version
- By a release-version and feature name
- By a release version, commit identifier and feature name

The user may also add the feature-mappings to the project to enable the quick inspection of which features have been changed per software release. While tis is an useful feature the process of mapping the features can be time-consuming work and developer guidance should be utilized when analysing the software structure.

When investigating issues from an issue tracking system the user may utilize J-Ace to investigate the changed files and features that are related to the issue. If there are changes that the issue doesn't describe the a potentially hazardous change has been identified. To enable the full potential of J-Ace it is required that the developers use the project-specific issue identifier pattern that links the commit to the issue every time commits are made.

The J-Ace provides a list of changed files, the amount of direct changes, and amount of classes that depend on the changed classes and amount of dependants per each level. The user may drill down on each changed file to

visually inspect the dependencies of a class in a dependency graph. While this is a good feature to understand the detail of the implementation a new requirement can be identified from the results of the case studies. The external *git diff* utility had to be used to gain better understanding of the actual changes that had been made on the file level. While the J-Ace does analyse the *diffs* and store the information in the database on a source-line level, the user interface does not have a feature to visualize this data.

The results of the case studies show that the use of file-level granularity in the analysing of the source code may result in high amount of false-positives. This can guide the quality assurance to wrong direction for example in cases where a change set has changed a large amount of files but in non-functional manner, for example by making only documentation changes. To circumvent this problem the Method-level granularity for the analysis should be implemented.

10 Conclusions

10.1 About this study

This study focused on solving the problem of prioritizing and guiding the testing efforts in a Java EE software development project. The J-Ace project accomplishes the main objectives that were set. The tool that was implemented may be used to gain a high-level insight of the changes made to the features of a project per release version and per a set of commits with the same identifier.

A non-foreseen benefit is found when analysing an unfamiliar project. The J-Ace may be used to gain knowledge of a previously unknown project and its internal structure and dependencies. While testing the tool certain open-source projects were cloned and analysed, for example the *ElasticSearch*, *Apache Hbase* and *Apache JClouds*. These projects use Maven as their build tool. The *Maven* is a popular build tool in which by default the version is defined with a standard pattern (*/project/version*). In an ideal case where the version definition is specified in the same file for the whole duration of the

project, it is possible for J-Ace to analyse the complete life-cycle of the project. The user may then visually inspect the changes that were made to the features of the project per version. This information can be used to gain understanding how software projects evolve and to identify the types of changes that are made during each phase of its life-cycle.

The J-Ace could be used in a real-life software project to gain better understanding of the necessary requirements for further development. The tool can already give a overview of the progress during the implementation phase of a project for the non-developers, such as project managers and quality assurance teams. The tools enables the identifying the nature of issues per type. For example when a specific feature is changed the set of actually changed files can be identified. This information may be used to better estimate the work needed for similar issues that are to be implemented in the future.

10.2 Future plans

The case studies revealed needs for features that do not currently exist in J-ACe but which could be beneficial to the end-user. These feature proposals are listed in Table 17.

Table 17 - Features envisioned for the future

Feature	Description
Diff View	A view that shows the changes that were performed on a file-level would enable the user to visually inspect the overall nature and quality of the changes.
Feature-level dependency View	A new view could be beneficial that would indicate a broader level of dependencies, i.e. the feature level dependencies instead of file level

	dependencies. The feature is feasible to implement as the dependency data is already being analysed on a finer level.
Method-level Granularity	The *Method* -level granularity feature should be implemented to accomplish a finer detail in the analysis and increase the accuracy of the results.
Timeline	The timeline should be configurable in the user interface to give user the possibility to show changes from a given time period.
Inspected Mark	A new feature should be introduced where the user may mark Commit IDs as inspected. After marking the commit it will be hidden in the views by default. The inspected commits could be shown in a separate view, or their visibility could be controlled by a flag.
Estimated Impact Set Analysis	While the J-Ace is a Actual Impact Set analyser it does collect a lot of data about the changes that have been done to the source code on a line-level granularity. This data could be used to help the developers in the determination of the estimated impact set. Example use-case: The developer enters a file that is estimated to be changed. The J-Ace collects the historical data of the file and

	determines the sets of files that have been changed in conjunction with the file that was entered.

10.3 Final words

The work hours for the implementation phase turned out to be larger than estimated. As the Euler diagram in chapter 4.6 indicates the combination of Fast and Good and Cheap may not be achieved. In this project the traits Fast and Cheap realised. Nevertheless the quality was attempted to be kept as high as possible with the limited resources that were available. The J-Ace tool consists out of over 6300 effective lines of Java-code. The source code is available in *GitHub*. It would be beneficial to gain community interest and support for further development of the project (see Appendix).

There is still room for improvements. The J-Ace could evolve into an Impact Analysis framework that hosts a multitude of different algorithms both for EIS and AIS phases. For example a developer could use the system during estimation phase to gain information about what parts of the system are usually changed in conjunction with the component is envisioned to be changed. The previous research (chapter 3.4) could be used as basis for implementing these algorithms.

References

Alfresco Wiki. Accessed 12th of November 2014. Retrieved from https://wiki.alfresco.com/wiki/Main_Page

Biggelaar W., 2014. Requirements Management in Real Life. Accessed 2th of July 2014. Retrieved From: www.processvision.nl/Training/RM in real life.pdf

Bohner S., Arnold R., 1996. Software Change Impact Analysis. Wiley-IEEE Computer Society Press

Bohner S., 2002. Extending Software Change Impact Analysis into COTS Components. Virginia Tech, Dept. of Computer Science. Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE, 175-182

Canfora G. Cerulo L., 2005. Impact Analysis by Mining Software and Change Request Repositories. Software Metrics, 2005. 11th IEEE International Symposium. DOI: 10.1109/METRICS.2005.28. Accessed 9th of May 2014. Retrieved From http://www.researchgate.net/profile/Luigi_Cerulo/publication/4175563_Impact_analysis_by_mining_software_and_change_request_repositories/file/e0b4951c98ff23ea92.pdf

Canfora G., Cerulo L., 2006. Fine Grained Indexing of Software Repositories to Support Impact Analysis. Proceedings of the 2006 international workshop on Mining software repositories. 105-111. DOI: 10.1145/1137983.1138009. Accessed 9th of May 2014. Retrieved From http://ifipwg213.org/sites/flosshub.org/files/105FineGrained.pdf

Context and Dependeny Injection for Java 1.1, 2014. Accessed on 12th of May 2014. Retrieved From https://jcp.org/aboutJava/communityprocess/final/jsr346/index.html

DevProd Report Revisited: Version Control Systems in 2013. Accessed 17th of November 2014. Retrieved from http://zeroturnaround.com/rebellabs/devprod-report-revisited-version-control-systems-in-2013/

Dustin E., Rashka J., Paul J., 2007. Automated Software testing. Addison-Wesley.

Enterprise JavaBeans. Accessed 28th of April 2014. Retrieved From http://en.wikipedia.org/wiki/Enterprise_JavaBeans

Enterprise Software. Accessed 25th of August 2013. Retrieved from http://en.wikipedia.org/wiki/Enterprise_software

Farley J., Crawford W., Malani P., Norman J., Gehtland J., 2005. Java Enterprise In a Nutshell. O'Reilly.

Glass R., 2001. Frequently Forgotten Fundamental Facts about Software Engineering. IEEE Software Magazine May 2001. Accessed 30th of April 2014. Retrieved from http://www.eng.auburn.edu/~hendrix/comp6710/readings/Forgotten_Funda

mentals_IEEE_Software_May_2001.pdf

Hevner A., March T., Park J., Ram S., 2004. Design Science in Information Systems Research. MIS Quarterly 28 (1), 75-105. Accessed 30th of April 2014. Retrieved From http://em.wtu.edu.cn/mis/jxkz/sjkx.pdf

Hibernate Lazy Loading. Accessed 24th of June 2014. Retrieved From http://stackoverflow.com/questions/10161165/jpa-which-implementations-support-lazy-loading-outside-transactions

Itkonen J., Mäntylä M., Lassenius C., 2007. Defect Detection Efficiency: Test Case Based vs. Exploratory Testing. Helsinki University of Technology, Software Business and Engineering Institute. DOI 10.1109/ESEM.2007.56. Accessed 30th of April 2004. Retrieved From https://wiki.aalto.fi/download/attachments/58925787/Itkonen_Juha_ESEM2007.pdf?version=1&modificationDate=1306496284000&api=v2

Java API for JSON Processing 1.0, 2014. Accessed 12th of May 2014. Retrieved From https://jcp.org/aboutJava/communityprocess/final/jsr353/index.html

Java EE Platform Specification. Accessed 15th of September 2013. Retrieved from https://java.net/projects/javaee-spec/pages/Home

Java Message Service 2.0, 2014. Accessed 12th of May 2014. Retrieved From https://jcp.org/aboutJava/communityprocess/final/jsr343/index.html

Johnson P., 2005. Impact Analysis - Organisational Views and Support Techniques. Licentiate's Thesis. Blekinge Institute of Technology. ISBN: 91-7295-059-5. Accessed 8th of may 2014. Retrieved from http://www.netlearning2002.org/fou/forskinfo.nsf/all/78f402765e403e73c1256ff70030a3ef/$file/licentiate_thesis_per_jonsson.pdf

Kagdi H., Gethers M., Poshyvanyk D., Collard M., 2010. Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. Reverse Engineering (WCRE), 2010 17th Working Conference. 119 – 128. DOI: 10.1109/WCRE.2010.21. Accessd on 9th of May 2014. Retrieved From http://www.cs.wm.edu/semeru/papers/WCRE2010_KagGet.pdf

Kaner C., Bach J., Pettichord B., 2011. Lessons Learned in Software Testing: A Context-Driven Approach. Wiley.

Lehnert S., 2011. A Review of Software Change Impact Analysis. Ilmenau University of Technology, Department of Software Systems / Process Informatics. URN: urn:nbn:de:gbv:ilm1-2011200618. Accessed 30th of April 2014. Retrieved from http://www.db-thueringen.de/servlets/DerivateServlet/Derivate-24546/ilm1-2011200618.pdf

Li B., Sun X., Leung H., Zhang S., 2012. A survey of code-based change impact analysis techniques. DOI: 10.1002/stvr.1475. Accessed 30th of April 2014. Retrieved from http://homes.cs.washington.edu/~szhang/pdf/cia-survey.pdf

Lindvall M., Komi-Sirviö S., Costa P., Seaman C., 2003, Embedded Software Maintenance, The University of Maryland. Accessed 28th of September 2014. Retrieved From: https://sw.csiac.org/get_pdf/CSIAC-347003-000.pdf

Lindvall M., Sandahl K., 1998, How Well do Experienced Software Developers Predict Software Change?, Journal of Systems and Software 1 (43) 1998

Mens T., Buckley J., Zenger M., Rashid A. 2004. Towards a Taxanomy of Software Evolution. Accessed 30th of April 2014. Retrieved From http://lampwww.epfl.ch/~zenger/papers/use03.pdf

Mikael Lindvall, Seija Komi-Sirviö, Patricia Costa and Carolyn Seaman

Moscow Method, 2014. Accessed on 10th of May 2014. Retrieved From http://en.wikipedia.org/wiki/MoSCoW_method

Myers G., Badgett T., Sandler C., 2011. The Art of Software Testing Third Edition. Wiley.

Peffers K., Tuunanen T., Gengler C., Rossi M., Hui W., Virtanen V., Bragge J, 2006. The Design Science Research Process: A Model for Producing and Presenting Information Systems Research. DESRIST, 86-104. Accessed 30th of April 2014. Retrieved from http://www.wrsc.org/sites/default/files/documents/000designscresearchproc_desrist_2006.pdf

Petrenko M., Rajlich V., 2009, Variable Granularity for Improving Precision of Impact Analysis, Wayne State University, Department of computer science. Accessed 17th of November 2014. Retrieved from http://www.cs.wayne.edu/~severe/publications/Petrenko.ICPC.2009.VariableGranularity.pdf

Rajlich V., 2012. Software Engineering: The current practice. CRC Press

Riley T., Goucher A., 2010, Beautiful Testing, O'Reilly Media Inc.

What is Enterprise Software. Accessed 9th of November 2013. Retrieved from http://www.perlmonks.org/?node_id=504043

Whittaker J., 2010, Exploratory Testing, Addison-Wesley.

Wonders of the J2EE Architecture. Accessed 9th of November 2013. Retrieved from http://aspalliance.com/1148_Wonders_of_the_J2EE_Architecture.2

Zimmermann T., Weißgerber P., Diehl S., Zeller A., 2005, IEEE Transactions on software engineering 6 (31), 429-445. Accessed 30th of April 2014. Retrieved from https://www.st.cs.uni-saarland.de/papers/icse2004/icse.pdf

Appendix – J-Ace project pages

The J-Ace project pages are hosted in GitHub. The source code, Issue-tracker and Wiki are available there.

URL: https://github.com/aironi/jace