Universidade do Minho
Escola de Engenharia

Mario Filipe de Melo Medeiros Fernandes Pinhal

An Integrated Environment for Software Assessments

November 2009

Universidade do Minho
Escola de Engenharia

Mario Filipe de Melo Medeiros Fernandes Pinhal

An Integrated Environment for Software Assessments

Master of Science in Informatics Dissertation

Supervisors:

Dr. ir. Joost Visser
The Software Improvement Group
Amsterdam, The Netherlands

Doutor Jose Bernardo Barros
Departamento de Informatica, Universidade do Minho
Braga, Portugal

November 2009

Universidade do Minho
Departamento de Informática

Master of Science in Informatics Dissertation

November 6, 2009

escola de engenharia

departamento de informática

An Integrated Environment for Software Assessments

Mário Filipe de Melo Medeiros Fernandes Pinhal
PG 13957

Supervisors:

Dr. ir. **Joost Visser**
The Software Improvement Group
Amsterdam, The Netherlands

Doutor **José Bernardo Barros**
Departamento de Informática, Universidade do Minho
Braga, Portugal

This thesis was written at the Software Improvement Group (Amsterdam, The Netherlands) with additional economic support of an *Erasmus Student Mobility for Placements*[1] grant in order to obtain the MSc. Degree in Informatics at Universidade do Minho.

[1] see http://ec.europa.eu/education/erasmus/doc894_en.htm

Acknowlegments

This dissertation would not have been a real fulfillment without the support and cooperation of all of those that directly or indirectly contributed to it. I owe everlasting gratefulness to many people who pleasantly involved themselves in helping me undertake this dissertation.

I am heartily thankful to my supervisor at SIG, Joost Visser, whose encouragement, guidance and support from the beginning to the final stages of my work, enabled me to keep on the right track and focused on the real issues of this thesis. My recognition also goes to José Bernardo Barros, from all the inspiration and clearness of mind, not only during this thesis work, but on all the many classes attended to the completion of my degree at Universidade do Minho.

In my daily work at Software Improvement Group (SIG), I was lucky to write my thesis in such an inspiring environment, cultivated by all those individuals from SIG staff. My words of thankfulness to all of SIG employees that happily engaged in the discussions, meetings and in the validation experiment, with the ones with whom I shared nice chats during my daily train trips to and from work or during lunch preparation at SIG. I couldn't have made it without all of you. In special: to Cora's by her support and laughs, to José Pedro Correia from helping me since my first days at SIG, always being available to brainstorm and discuss, to Tiago Alves from his input and revisions on this thesis report, and to Rob van der Leek and Wander Grevink for the technical support and the patience for sitting down with me to discuss and address some of the problems that this thesis focuses in. Also to Leo Makkinje and Gerard Kok, for their patience, availability and mood with which they always attended to my requests.

My recognition also to the University of Minho and to the Informatics Department, for giving me the best conditions to pursue my studies, to the several professors that will always remain an inspiration to me, and to the GRI for helping me in pursuing such path with my studies abroad in the Netherlands.

Beyond Informatics and this thesis, I am most grateful to my parents not only for supporting me and providing me the means for my education, but also for all the care and love, during my whole life and academic path. My warm thanks to my friends Milena Tsvetkova, Antionetta Stefanova and Emma Martinez for hosting me while moving from place to place, always receiving me with "open warms" and a smile.

My work would definitely not have been possible without the moral support, encouragement and care of Aliki Tzatha. She was more than a partner and companion, a true friend, inspiring me in so many ways every day that passes.

Finally, I would like to thank everybody else, friends and colleagues, who were important to the successful realization of this thesis, as well as expressing my apology that I could not mention personally one by one.

Abstract

The SIG is a consultancy firm offering assessments of quality and maintainability of software to its clients. To execute such assessments, SIG performs static source code analysis over the software systems of its clients. For this purpose, SIG uses a set of in-house developed tools to extract facts and metrics, thus obtaining insight into these software systems.

Due to the company's fast growth and to the increase of the volume and complexity of the company's software, SIG seeks to further professionalise and streamline the delivery of its services by standardising its software analysis process. Rather then simply having a collection of tools, SIG's aim was to integrate them into an encompassing system – an *Integrated Environment*, through the development of a Graphical User Interface (GUI). By doing so, the tools can be aligned more closely with the current Standard Operating Procedures (SOP) followed during the software assessments, contributing to the standardisation and improvement in the coherence, consistency and effectiveness of the analysis process.

In this thesis, the tools of SIG and its activities are studied, in order to obtain knowledge about the system and draw the requirements for our integration task. Some effort was first spent to automate the whole process of software analysis, giving particular attention to the configuration phase. A tool was created in order to assist the inspection of software system archives and classify their contents, in order to define the scope of the analysis. We automatically detected the used technologies, separating test code from source code, verifying also the presence of files which are not relevant for the analysis process (such as generated source code).

To find a GUI technology to fit our requirements, we defined a list of criteria and performed a comparative research study over currently available Desktop Application and Rich Internet Application GUI technologies. We decided to use the Apache Wicket framework to build our GUI application.

The research carried out and the introduction of the developed Integrated Environment, proved to be valuable, automating and fully supporting the software analysis process and the assessments conducted by SIG. Furthermore, this work also helped to support the Software Certification service at SIG and the Monitoring of Open Source Systems project and subsequent related research work.

Keywords Integrated Systems, Tool Automation, Tool Integration, Graphical User Interfaces, Patterns Detection, Interaction Design, Software Analysis

Abstracto

A empresa Software Improvement Group (SIG) realiza avaliações à qualidade e manutenção de sistemas de *software* dos seus clientes, recorrendo à análise estática do código fonte desses mesmos sistemas de *software*. Para tal, a SIG desenvolve e mantém um conjunto de ferramentas para a extracção de factos e métricas, no sentido de obter mais detalhe e compreensão nas peças de *software* analisadas.

Devido ao rápido crescimento da empresa e ao aumento do volume e complexidade do *software* produzido pela empresa, a SIG procura profissionalizar a prestação dos seus serviços, normalizando o processo de análise de *software*. Em vez de dispor de apenas uma colecção de ferramentas, a SIG pretende integrar as suas ferramentas num sistema único - um Ambiente Integrado, através do desenvolvimento de uma *Interface* Gráfica com os seus utilizadores. Assim sendo, as ferramentas são alinhadas com os Procedimentos Operacionais Padrão seguidos durante as avaliações de *software*, contribuindo assim para a normalização e aumento da coerência, consistência e eficácia do processo de análise.

Nesta dissertação, as ferramentas da SIG e as suas actividades foram estudadas, para se obter conhecimento acerca do sistema e definir requisitos para a integração do sistema. Os primeiros esforços foram primeiro gastos em automatizar todo o processo de análise de *software*, dando particular atenção à fase de configuração. Uma ferramenta foi criada para assistir à inspecção de arquivos de sistemas de *software* e classificar os seus conteúdos, de forma a definir o alcance e configuração do processo de análise. Foram automaticamente detectadas as tecnologias usadas, separando código fonte teste de código fonte de produção, e verificada também a presença de ficheiros não relevantes (como por exemplo código fonte gerado).

Para encontrar uma tecnologia que cumprisse os nossos requisitos, definimos uma lista de critérios e realizamos um estudo comparativo sobre tecnologias para desenvolvimento de Aplicações *Desktop* e Web. Decidimos usar a *framework* Apache Wicket para desenvolver a nossa *Interface* Gráfica com o utilizador.

A investigação conduzida e a introdução do Ambiente Integrado desenvolvido, automatizaram e assistiram ao melhoramento do processo de análise de *software* conduzido a par das avaliaçoes conduzidas pela SIG. Este trabalho também contribuiu como suporte para um novo serviço de certificação de *software* assim como em projectos investigação tais como o de Monitorização de Sistemas *Open Source*.

Palavras Chave: Integração de Sistemas, Automatização de Ferramentas, Integração de Ferramentas, Interfaces Gráficos com o Utilizador, Detecção de Padrões, *Design* de Interacção, Análise de Software

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we start by introducing the Software Improvement Group – the company at which this dissertation was conducted – and its activities (1.1). We follow on by presenting our motivation (1.2), our objectives and approach (1.3), and we conclude with a section about the structure of this dissertation (1.4).

1.1 The Software Improvement Group

The Software Improvement Group (SIG) is a consultancy firm which focuses its business and services on performing assessments of quality and maintainability on its customers' software. SIG started as a spin-off of the Centrum voor Wiskunde & Informatica[1] (the Dutch National Research Institute for Mathematics and Computer Science, commonly known as CWI). In the late 1990s, the CWI was conducting research into how to map out software quality using source code analyses. One of CWI's research partners (ABN Amro) was interested in putting the research results into practice and SIG was then founded.

SIG offers IT management consultancy services to its customers where source code analysis plays a major role. These services are delivered in two basic forms: Software Risk Assessment (SRA) and Software Monitor (SM). In an SRA, a single snapshot of a system is investigated in-depth. In SM, the evolution of the quality of a software system is tracked through time.

To deliver these services, the consultants of SIG carry out software assessment activities, where information is gathered from multiple sources: source code, documentation, interviews, history of benchmark repository, among others. This information is subsequently combined and analysed in order to arrive at quality appraisals and strategic recommendations which are delivered to the client.

Furthermore, SIG has been working in order to expand its range of services. During the time this thesis is written, a new service was launched: Software Product Certification (SPC) – with the goal of certifying the quality of software systems. SIG also maintains a benchmark of software quality metrics and appraisals, providing a frame of reference for evaluating software quality, allowing comparison

[1]For more information, consult http://www.cwi.nl/

with peers in industry, technology and architecture.

On an annual basis, the SIG analyses over 50 software systems for corporate and institutional clients.

1.2 Motivation

SIG is constantly aiming to further professionalise and streamline the delivery of its services. We have analysed known bottlenecks and elaborated them into a list of potential areas for improvement. The list is as follows:

Lack of uniformity in generation of the deliverables The current SAT is currently composed by several tools. Although there are periodic stable releases of these tools, the developers – technical consultants – still have to compile and install them locally. This is labour intensive and error prone, sometimes resulting in incoherent generation of deliverables and difficult trace of such incoherences.

Lack of Traceability and Repeatability Traceability is difficult to ensure in what concerns the deliverables of the system. Different technical analysts follow different procedures, sometimes leading to incomplete or inconsistent deliverables. This means that although there is a record that a certain deliverable was created and sent to the client, there is no insurance that the reproduction of its generation will produce the same exact deliverable.

Lack of data consolidation among deliverables There is no current data consolidation among deliverables. This happens due to the above mentioned points, implying problematic validation of SIG's Quality Model, and also problematic comparison across systems with the same set of technologies and characteristics of SIG's Benchmark.

Rework A lot of rework is done at the SIG, mostly in inspecting the code base of a software system and in the refinement of its scope configuration (see Chapter 4). There is no tool that supports the creation of this scope configuration before executing the analysis process, and there is also no form to validate such configuration. Most of the time, this implies that the analysis is performed several times until the desired configuration is obtained. This is a very inefficient procedure, leading to a considerable waste of time and resources.

Learning effort and cost There is a big learning effort in what concerns the Software Analysis procedure, its toolkit and the current associated Standard Operating Procedures (SOPs). New employees that arrive at the SIG have access to documentation about the followed SOPs and about the SAT that is used to run the analyses, but cannot easily understand the workflow of the whole process and the use of the diverse tools.

Errors / Incompleteness Due to the lack of support of configuration, the scope of an analysis is not always correct. Some files that should be excluded aren't, generated code is not identified and so analysed as production code, among other situations.

SIG is seeking to integrate its various tools it has been developing into an encompassing system. Rather then simply having a collection of tools, SIG seeks to provide its various types of users with an Integrated Environment, in order to support the software quality and risk assessments conducted by SIG employees and provide them an easy, clear and intuitive interface to the current functionality of SIG's collection of tools.

1.3 Objectives and Approach

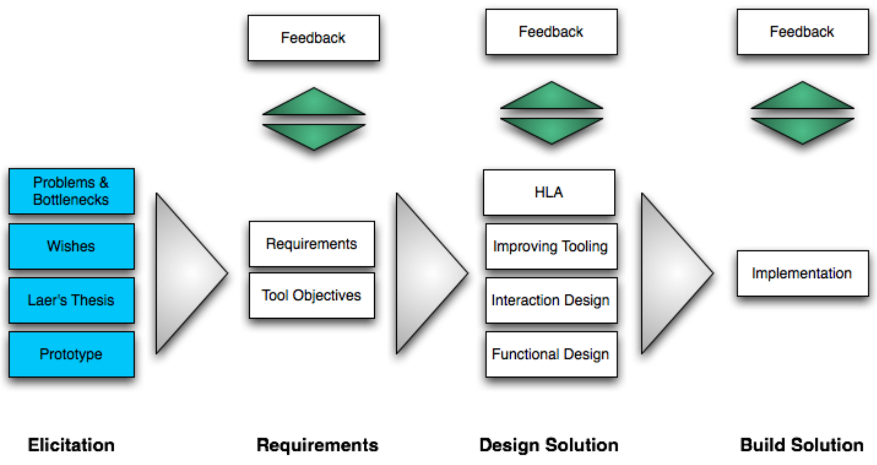


Figure 1.1: Approach

The main objective of this project is to reduce the above presented problems through the creation of an Integrated Assessments Environment. Our approach is described by figure 1.1.

First, we perform an elicitation research study over SIG activities. We use several elicitation techniques and we gather the necessary information about SIG services, how these services are delivered and what tooling supports their delivery. This allows us to obtain knowledge, context and comprehension, necessary for analysing the current situation and related problems and bottlenecks reported by SIG staff.

Additionally, we take also in consideration SIG's objectives and users preferences and wishes. We then define a concrete list of requirements and objectives for the development of our integrated environment. We approach these requirements and work first on improving tool support, in order to automate the full software analysis procedure performed at SIG.

Having automated this procedure, we do interaction design and we perform a comparative research study on currently available GUI technologies in order to select the technology to build our Integrated Environment. Finally, we build a GUI that integrates the tooling available at SIG, achieving our goal of producing an Integrated Assessment Environment.

At each stage, several feedback meetings took place as validation from the final users of the Integrated Environment (IE) .

1.4 Structure of this dissertation

The structure of this thesis goes as follows:

Elicitation (Chapter 2) We present how we acquire and elicit the necessary information about SIG, its services and activities and we also define the requirements for our Integrated Environment (IE).

Related Work (Chapter 3) We present related work on Integrated Environments and Scientific Workflow areas.

Configuration (Chapter 4) We present a tool that we built in order to automate the workflow and assist SIG employees in the configuration of an analyses.

Graphical User Interface (Chapter 5) We document the design, the selection of technology and the architecture of the implemented GUI.

Validation (Chapter 6) In this chapter, we document the design and execution of a small validation experiment to evaluate the usefulness of the configuration procedure using the GUI.

Conclusions (Chapter 7) Finally we present our conclusions, a small discussion section and future work directions.

Chapter 2

Elicitation

To be able to obtain complete understanding about the system to be integrated, we conducted a research study on SIG's activities, provided services and set of tools and technologies, in order to gather a list of requirements for our IE.

This chapter first describes how the information was acquired, by describing the techniques (2.1) and used sources (2.2) that we used in our elicitation research. We follow on presenting and describing SIG services (2.3), the software analysis performed at SIG (2.4) and the tools that exist to support the delivery (2.5) of these services. We then formulate a list of requirements that guided us in our work and research for the rest of this thesis (2.6). Finally we describe how we addressed these requirements and which solutions we found to fulfil them (2.7).

2.1 Elicitation techniques

To understand what the different services provided by SIG are and how these are conducted, we used several elicitation techniques. We used the same techniques as in [12] – interviewing, apprenticing and document archæology. In addition, we used a Business Process Modelling Notation (BPMN) model to visualise the SRA process, and we performed Source Code inspection as well, to be able to understand the direct connection between the diverse tools of the SAT.

In **Document Archæology** the elicitor tries to get insight in the process on the basis of existing documentation[12]. In the case of the SIG, these were SOP documents, project specific documents such as deliverables, notes, findings, and as well general documents as templates, questionnaires, procedures, etc.

Apprenticing refers to the action of learning the work as it is done by the expert[12]. The apprentice gets a detailed view of the process by observing it while it is being done. It is not the purpose of the apprentice to learn the job so one can execute it, but it offers the most concrete insight in a process. Disadvantages of this technique are the concrete view of the process and thus the lack of generalisation. It is hard for the apprentice to judge how representative the project is for all projects in the organisation. In comparison to other techniques, apprenticing usually takes more time[12].

Interviews are quick and easy to set up. They provide the elicitor with a good entry point for the process but it are particularly hard to get a full and profound insight. Interviewing has also some disadvantages. The interviewee will present an abstract depiction of the subject to simplify his or her talk. The fact that this happens in a verbal way, increases the amount of misunderstandings. Furthermore an interviewee can forget things or can be biassed. Although we used interviews, we tried to cover these disadvantages of biassed interviewees and abstractions by asking for examples. Furthermore we used sample documents from earlier projects to track and check the interviews.

By performing **Source Code Inspection** we are able to verify the tool connections and to check which data models are used and shared by these tools. Furthermore, it gives us more insight about how things are actually done, rather than the more abstract explanation provided by documents or by the explanation given in interviews.

2.2 Sources

In our elicitation research, we used several sources. We performed several interviews and sessions with different SIG's employees about SIG's services, the SAT and the associated SOPs. We attended a session with a client, to get concrete example and further contextualise the information that was being collected and elicited.

We sat down and apprenticed several details about the software analysis and about the use of the related SAT tools among other development aspects. Furthermore, we also inspected the source code of the tools that compose the SAT. These gave us even more insight into the relation between the tools, the data models behind them, and their specifications and limitations.

Aside from the previous described sources, we used several internal SIG documents such as project reports, contracts with clients, session templates, SOP guidelines documents, and also scientific published articles (such as [6], [8] and [4]).

2.3 Services delivered by SIG

2.3.1 Software Risk Assessment

A Software Risk Assessment is an independent investigation of the technical quality and risks of a software system, based among others on automated source code analysis. The main goal is to provide the client with concrete recommendations to minimise the risks in the maintainability of their system(s).

The overall approach of SIG to the SRA is depicted in Figure 2.1. In addition to the data derived from the source code analysis (section 2.4), interviews are conducted with various kinds of stakeholders in the system to obtain technical and strategic information. This includes business goals, design information, process information, history, external architecture (relation to other systems), operational indicators (how many users, transactions, problems), etc.

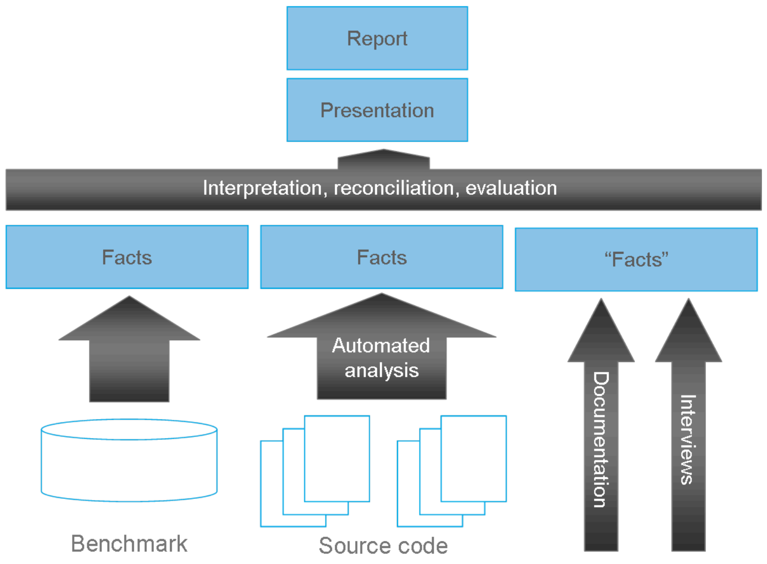


Figure 2.1: Software Risk Assessment Methodology

If reliable documentation is available, a global or more profound review of this documentation is performed to obtain information about the (intended) architecture, functional and technical design, etc. Also, data about previous systems analysed by SIG is collected and recorded in a benchmark repository. This information is used to compare the system currently under study to similar systems.

The SRA is finalised in two separate parts. First, a final presentation is given to the client. In this presentation the findings and conclusions are presented. During this presentation there is room for questions and discussion. After the final presentation a formal report is sent to the client. In this report the relevant findings, results, interpretation and risks of migration identified will be written down. In the management summary of the report the most important risks, conclusions and recommendations are presented.

2.3.2 Software Monitor and Software Portfolio Monitor

The Software Monitor (SM)/Software Portfolio Monitor (SPM) is a service of SIG that gives insight about the technical quality of a software system code base by measuring the system's software quality on a regular basis. This enables organisations to manage their software development, processes and software maintenance based on facts: Fact Based ICT Management.

As shown in Figure 2.2, the SM service offers relevant information on different management levels. The client decides who will receive which reports. The SM is being implemented in order to attain various objectives. In any case there is the need for transparency of technical quality in software systems. Below are some of the situations where SIG's SM is typically used:

- In the continuous assessment of the performance of (external) software de-

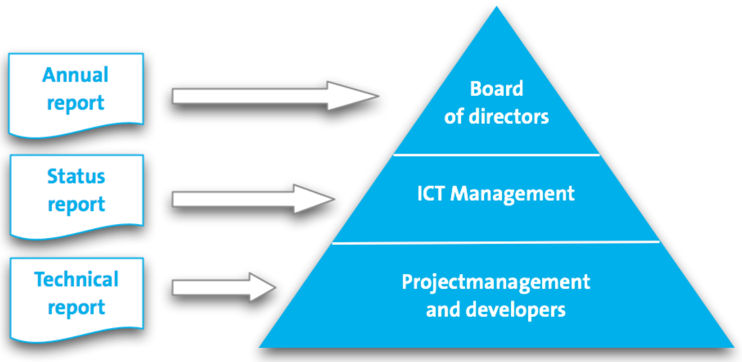


Figure 2.2: Software Monitor - Delivery of reports

velopment teams.
- As part of Service Level Agreements.
- To guarantee quality or mitigate risks of large strategic software development projects.
- The assessment of a portfolio of software systems.
- As a basis for certification of systems.

The SM enables organisations to monitor technical aspects of their software. It is possible to monitor size, complexity, architectural issues, coding standards and technical quality specifications. Generally there are two different categories of measurements (also called metrics):

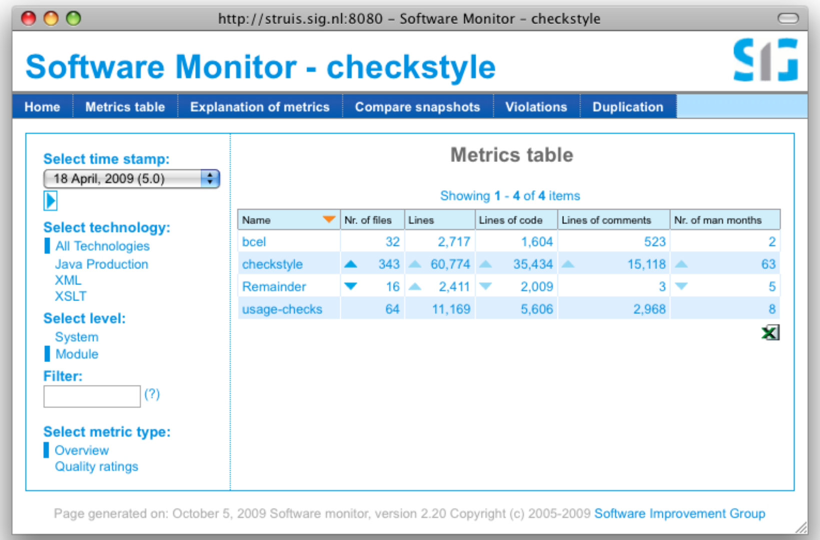
- Informative measurements: used to get a clear picture of the system and track the development of the system over time. An example of such measurement are for instance the number of source code lines and the number of modified source code lines per week/month.
- Quality measurements: used to obtain detail on the quality of a system and the parts that compose it. This second category of metrics does have a maximum or optimal value. For these metrics, SIG might determine desirable, optimal or maximum values. Duplication is an example of such quality measurement.

All measurements are taken using automated source code analyses (section 2.4). The result of the measurements is shown on different abstraction levels (for example; system level, package level or file level). This is shown in Figure 2.3.

Unique to the SM is that it is possible to follow the features of the application over time (see figure 2.4). The development of quality aspects during a period of time provides an overview of where improvements are being made. It also becomes clear the result of the work that was carried out during a certain period of time.

2.3.3 Software Product Certification

The purpose of a certificate is to give written assurance that the product has been evaluated by a qualified party in a neutral and repeatable way, and has thereby



Software Monitor - checkstyle

Select time stamp: 18 April, 2009 (5.0)

Select technology: All Technologies, Java Production, XML, XSLT

Select level: System, Module

Filter: (?)

Select metric type: Overview, Quality ratings

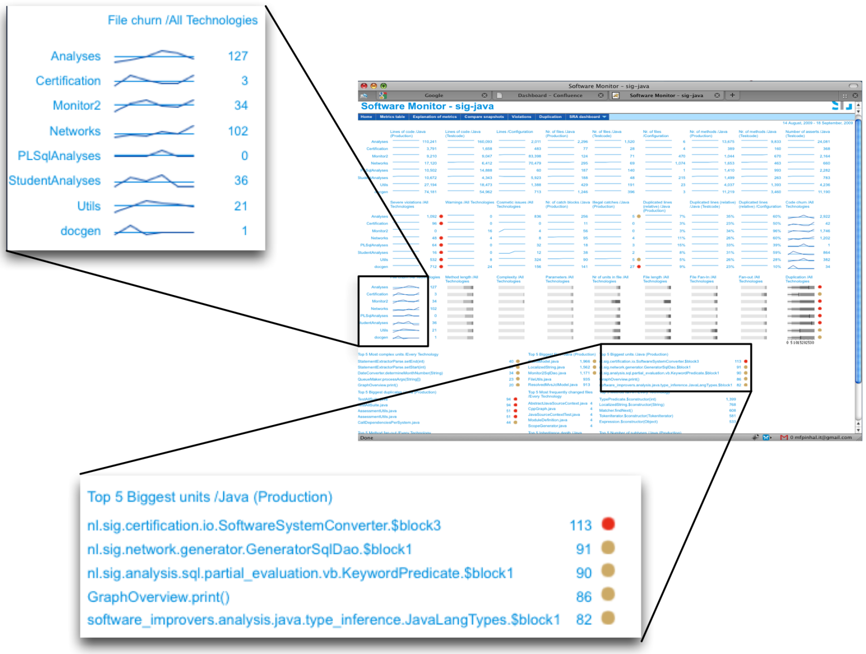
Metrics table

Showing 1 - 4 of 4 items

Name	Nr. of files	Lines	Lines of code	Lines of comments	Nr. of man months
bcel	32	2,717	1,604	523	2
checkstyle	▲ 343	▲ 60,774	▲ 35,434	▲ 15,118	▲ 63
Remainder	▼ 16	▲ 2,411	▼ 2,009	3	▼ 5
usage-checks	64	11,169	5,606	2,968	8

Page generated on: October 5, 2009 Software monitor, version 2.20 Copyright (c) 2005-2009 Software Improvement Group

Figure 2.3: Software Monitor - Metrics table page



File churn /All Technologies

Analyses	127
Certification	3
Monitor2	34
Networks	102
PLSqlAnalyses	0
StudentAnalyses	36
Utils	21
docgen	1

Top 5 Biggest units /Java (Production)

nl.sig.certification.io.SoftwareSystemConverter.$block3	113 ●
nl.sig.network.generator.GeneratorSqlDao.$block1	91 ●
nl.sig.analysis.sql.partial_evaluation.vb.KeywordPredicate.$block1	90 ●
GraphOverview.print()	86 ●
software_improvers.analysis.java.type_inference.JavaLangTypes.$block1	82 ●

Figure 2.4: Software Monitor - Dashboard page (main page)

been shown to meet relevant evaluation criteria. A certificate is an evidence of quality, gives confidence, and provides an objective basis for agreement and comparison. In collaboration with TÜV Informationstechnik (TÜViT), SIG offers the certification of the maintainability of software products as defined by the international standard ISO/IEC 9126.

To determine eligibility for certification, the maintainability of a software product is measured by automated source code analysis. The measurement model is based on ISO/IEC 9126, draws on extensive worldwide scientific research and has been developed and validated within SIG's assessment practice (see section 2.4 below).

The evaluation process starts by the submission of the source code of the software product and a high-level product description to the evaluation laboratory of SIG. These are evaluated by SIG using automated source code analysis and document review. Products are rated on a scale of 1 to 5 stars, and only products with 3 stars or more are eligible for certification. Based on the measurements and ratings, SIG prepares an evaluation report which is supplied to the TÜViT certification body. Based on the report, TÜViT awards a certificate to the software product and assigns the quality mark "TÜViT Trusted Product Maintainability".

2.4 How does SIG perform Software Analysis

To perform software analysis, all source code files of a system are submitted to static source code analysis to extract metrics, dependency and duplication information, coding standard violations, and other facts. Source code artifacts include not only the program files but also test code, configuration files, structured documentation, data models, etc.

For fact collection from system source code and other source artifacts, a SIG analyst will deploy the SAT for software analysis. If necessary, this toolkit will be extended or adapted by the developers team.

The software is also checked against common engineering practices, like described in the regulations of the Software Engineering Body of Knowledge[Web 23]. Depending on the technology used, SIG can furthermore check coding standards on a more detailed level.

SIG has developed a model for software product quality, which provides an operationalisation of the ISO/IEC 9126 International Standard for Software Product Quality [8]. SIG's software product quality model is not the only instrument for arriving at quality judgements in the SRA process. The model is intended to be a semi-structured core of the entire process. It does not cover the entire process, and it does not intend to discharge the SRA consultant and the SRA analyst from carrying out a conscientious and creative assessment. Part 1 of the ISO/IEC 9126 standard contains a model that dissects the notion of software product quality into six characteristics, which are further subdivided into a total of 27 sub-characteristics. The SRAs carried out by SIG focus on technical rather than functional quality of software systems. This means that only a subset of the software quality sub-characteristics of the ISO model is relevant. Figure 2.5 shows this breakdown.

To make the ISO/IEC 9126 quality model operational, a mapping can be established from source code observations to system properties, and finally to quality sub-characteristics. This is shown in Figure 2.6.

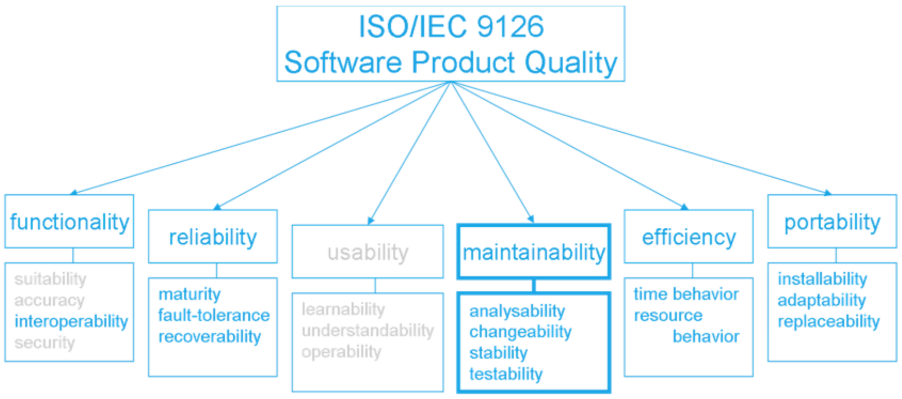
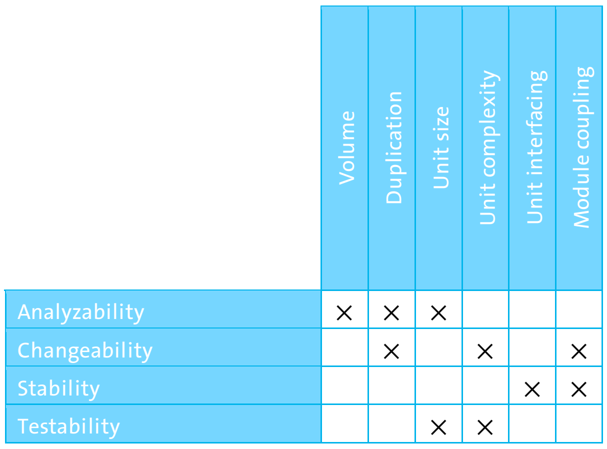


Figure 2.5: The SRA process focuses on the technical aspect of the software product, in particular on its maintainability.



	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling
Analyzability	×	×	×			
Changeability		×		×		×
Stability					×	×
Testability			×	×		

Figure 2.6: Relationship between system properties and a quality sub-characteristics

2.5 Which tools and assets support SIG service delivery

2.5.1 Software Analysis Toolkit (SAT)

The Software Analysis Toolkit is the name attributed to set of tools that are developed, maintained and used by SIG employees for static source code analysis. Among the SAT there are tools to configure, execute and visualise the analyses

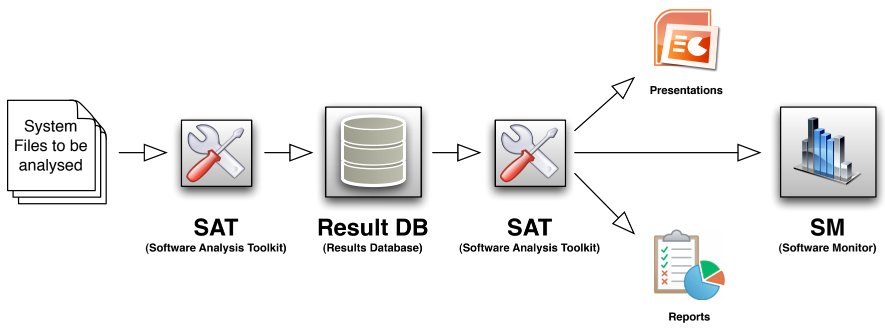

System Files to be analysed
SAT
(Software Analysis Toolkit)
Result DB
(Results Database)
SAT
(Software Analysis Toolkit)
Presentations
Reports
SM
(Software Monitor)

Figure 2.7: Source code analysis at SIG

of a system. These tools were developed in Java, having a simple Command line interface (CLI) for their usage.

Typically, as Figure 2.7 illustrates, a SIG analyst follows the SOP and executes the required tools in order to obtain the desired deliverable. Such situations are for instance running the analyses, generate reports or presentations, configure, create and deploy the SM of a software system. At each step of the chain, the analyst also checks for errors and warnings that are registered in log files created by the different tools.

Since these tools are currently and constantly under development, we should be concerned about how they evolve and what procedures and implications derive from their evolution.

2.5.2 Software Monitor

The Software Monitor provides access to the facts and metrics collected during the software analysis process conducted by SIG analysts. It is basically an interface to the collected data through a website, that is linked with a database where the metrics and facts were previously stored during the automated source code analysis procedure.

It is used by the analysts and consultants in order to the visualise the result of the source code analysis of a system, and as well by the clients of a SM/SPM project. As shown in Figure 2.7, the analyst recurs to the SAT for its generation and deployment.

2.6 Requirements

In this section we present the list of requirements derived during the elicitation process as described in this chapter. We group these by common properties which are normally a quality indicator of a software system.

2.6.1 Standardisation

Installation and usage of Tools To prevent human errors and inconsistencies, the installation and usage of the SAT and its tools should be standardised. Instead of the local compilation of sources and installation of these, they should be installed after a release of the SAT, and be made accessible to its users. Previous versions should as well be available as well to once again, prevent the same problems.

Scope Configuration The configuration of the analyses of a software system should be assisted and performed in a standard way. On the contrary to the current situation where the configuration procedure is inefficient and performed differently by different analysts, our integrated environment should assist all the steps of this procedure.

Workflow The workflow of the whole analyses process should be standardised, so that the steps are executed in conformation with the current SOPs. The GUI should as well provide the user with a clear overview of its overall progress.

2.6.2 Traceability and Repeatability

It should be possible to trace a deliverable (such as for instance a dependency call graph) till the original source code and analyses set. By achieving traceability of every deliverable, by its turn, also ensures repeatability, having all the necessary elements to reproduce the same analyses and generation of the same deliverables.

2.6.3 Automation

All the processes that imply manual workflow and combination of tools should be automated. Whenever possible, manual work should be eliminated unless there are reasons for explicitly requiring the attention of the user. The generation of deliverables should be as easy as possible, no longer having the user the need to combine tools and adapt its intermediate results in order to achieve the desired deliverable.

2.6.4 Consistency

The execution of the analyses over a code base of a software system should be done in a consistent way, as well as the generation of deliverables based on these analyses. Ensuring standardisation in the installation and usage of the SAT also contributes to the consistency of the analyses and its associated deliverables. Our integrated environment should ensure that there are no incorrect combinations of tools.

2.6.5 Integration

The tools of the SAT should be integrated by our GUI application. Furthermore, as it can be seen in Figure 2.8, the GUI should also integrate a tool to assist the configuration. The functionality of the diverse tools of the SAT should be linked by a backend, and a frontend should provide the access and allow the use of the diverse tools of the toolkit.

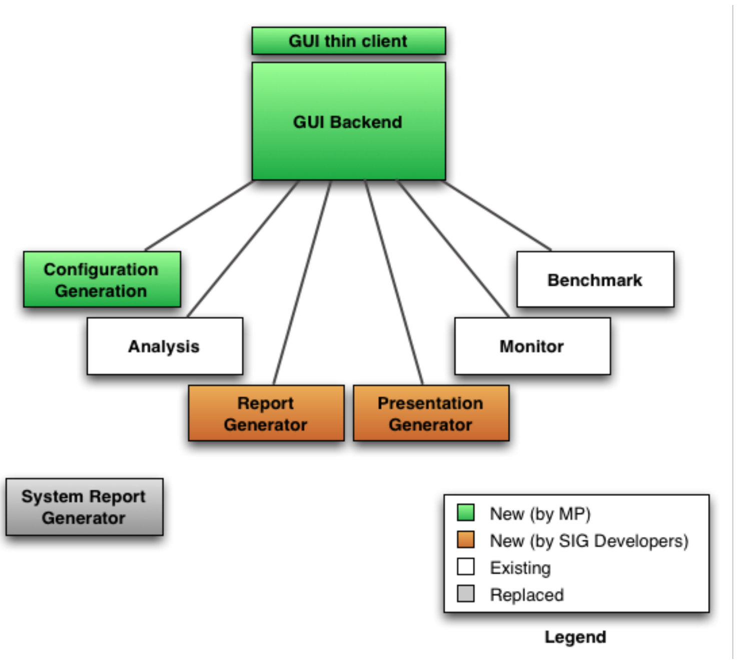


Figure 2.8: System Integration
SAT tools and new configuration generation tool are integrated by the GUI

2.7 Addressing the requirements

Here we briefly describe what actions were taken, in collaboration with the SIG staff, in order to meet the list of requirements above mentioned.

In what concerns the SAT, we ensured both its correct installation and its usage. The installation of the tools that compose the SAT is now done in a standard way. The tools are stored in a shared location where SIG employees can access to use them. All tools are installed as a bundle, in order to attain the combination of tools with compatible versions.

With the introduction of the configuration tool (that is described in detail in section 4), we ensured that the configuration of the analyses is done in a standard way in conformation with the related SOP.

For obtaining traceability and repeatability, we now ensure that every analyses set is now identified with the version of the SAT that was used, that the files of the software system over which the analyses are run cannot be changed, and we also keep a *timestamp* of its creation. The same happens with the deliverables.

We created an Integrated Environment (IE) where we integrate all the tools that compose it through the development of a GUI to the SAT. The IE guarantees that the usage of the SAT by SIG employees is tightly coupled with the designated SOP for source code analysis.

In the remainder of this thesis we focus mainly on satisfying the Integration requirement. In chapter 4 we focus on the integration of a tool to assist the scope configuration of the analyses, and at chapter 5 in the integration of the other tools through the introduction of the GUI as an interface to the SAT.

Chapter 3

Related Work

In this chapter, we briefly present the state of the art in the topics that surround this thesis final purpose of an *Integrated Environment for Software Assessments*. We start by presenting previous relevant work at SIG (3.1). We then introduce the reader to the topics of Tool Automation, Tool Integration and Integrated Environments within the context of Software Engineering Environments (3.2). We relate our work with Scientific Workflow Systems (3.3) and end the section by presenting related work about Graphical User Interfaces (3.4).

3.1 At the Software Improvement Group

In his recent work [12], van Laer – previous master student intern at SIG – focuses on modelling and improving the SRA conducted at the SIG. He first used several elicitation techniques to get insight into SIG's SRA. He created a business model of the SRA using Business Process Modelling Notation (BPMN). Then, he assessed SIG's SRA with a generic audit approach, identifying its weaknesses, problems and bottlenecks. Furthermore he also performed a research on Rich Internet Application (RIA) technologies and built a GUI application prototype to support the SRA and optimise its process.

In our work, we used the business model created by van Laer to reduce our learning time in understanding the SRA. We used this model in interviews, while inquiring the SAT and as a validation of our findings, as it was when van Laer was still building it. The identified bottlenecks helped us also in obtaining insight into which parts of the process needed to be assisted or optimised. These were good pointers which we used while defining the requirements for our integration and automation task. While van Laer focus is on modelling and assessing the SRA, identifying its problems and bottlenecks, our work was focused on finding a solution for part of these problems and others that we found during our elicitation research.

In comparison to van Laer's work, we went deeper on the requirements definition of our GUI application, since some aspects had not been taken into account in his research. We will go back to this aspect of van Laer's work at section 3.4 further in this chapter.

3.2 Tool Integration and Tool Automation

Tool integration remains a lively research topic, and more appropriate and fundamental research questions need to be debated posed and answered by the software engineering community [17]. In [16], Wicks created an annotated bibliography about Tool Integration within software engineering environments, presenting a reviewed list of relevant literature over the past decades. In [17], Wicks and Dewar analyse this bibliography to examine the background, context and scope of tool integration, and propose an empirical manifesto for future research.

Wicks and Dewar define “Tool integration concerns the techniques used to form coalitions of tools to provide an environment that supports some, or all, of the activities within a software engineering process”. In the same paper, Wicks and Dewar identify from their case study that business decisions and goals are an important driver in the adoption of tool integration solutions. Such is the scope of our research, as stated in Chapter 1. Wicks and Dewar further summarise that “the desire to integrate tools within an encompassing software engineering environment, is driven by the belief that derived benefits will accrue, through automation, with consequent productivity and quality improvements”.

Among their manifesto, Wicks and Dewar point out the fact that in the literature “there is little evidence of what problems they are alleged to solve and how successful they are”. We take these remarks into consideration and try to contextualise and provide answers to the posed above questions. We clearly stated which problems there were in the beginning, and how effectively we solved them (see Chapter 7). More specifically, we also performed a experimental validation to assess the developed GUI application (see Chapter 6).

In [14], Wasserman categorises the different types of tool integration. From these we performed Platform integration, Presentation integration and Process integration, leaving out Data integration and Control integration.

3.3 Workflow and Interaction Design

Recent years have seen a dramatic increase in research and development of scientific workflow systems [10]. Such systems promise to make scientists more productive by automating data-driven and compute intensive analyses such as the ones performed by SIG. In [10], a desiderata is identified focusing on enabling scientists to model and design the worflows they wish to automate themselves. In our work, we also design and automate our workflow, in order to link our analysis software tools with the current SOP followed during the software assessments at SIG. From these desiderata, we found relevant to our case well-formedness, predictability, recordability and reportability, that we use as criteria elements to define our requirements further in Section 2.6 of the of Chapter 2.

3.4 Rich Internet Applications vs. Desktop Applications

In [12], van Laer includes a comparison of GUI technologies, and decided to use a RIA to built his prototype. However, he does not focus on the direct implications of a choice of a RIA over a Desktop Application, being this worth of investigation. Also, it was not considered the use of other technologies and libraries to provide a richer interface. This is an important factor on this research – the usability and ease of use of the User Interface (UI) components that will compose our GUI application. We perform a more broad comparative research on GUI technologies, in order to include as well Desktop GUI technologies and ended up choosing a different one in order to best satisfy our requirements.

Chapter 4

Scope Configuration

The most relevant bottleneck of the current SOP for the software analysis performed by SIG is the configuration phase. Before executing the SAT over the source code files of a software system, it is first necessary to configure how the SAT will process its files.

The task is a rather important one. Software systems can contain thousands of files written in different programming languages. We need to assure that all the files of a software system are correctly classified in our configuration, so that the intended analyses are performed over the intended files. When a produced configuration for a given software system is incorrect, the analysis process can fail or generate incorrect and inconsistent results, that may even not be noticeable at first sight.

It is therefore a challenge to find ways to assist this configuration step, to automate the classification of the files of a software system and the detection of ambiguous files. At this chapter we document the creation of a tool to inspect the contents of a software system source code folder, classifying its contents and generating a configuration in a completely automatic manner.

Currently, there are two configuration formats at SIG for the definition of a Scope configuration. The first, using Spring[Web 18], and the latter using a simple eXtensible Markup Language (XML) file with the support of JAXB[Web 20] RI[Web 21]. The second configuration format is replacing the first, due to its conciseness, clearness and simplicity. Therefore we model our tool to use the latter format. This configuration format is referred to as *scope* configuration - since it defines the scope of the analysis.

The *scope* configuration (see Figure 4.1) is composed by the following information:

Software System information: the name of the system, the client name, the project code, the path where the source code folder is located.

Technologies information: which programming languages compose the software system and its associated set of files. Furthermore, there is also the possibility to define subsets for each programming language. Typically, this occurs dividing source code files from test code files.

Listing 4.1: Scope.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<system projectCode="test" name="test"
        xsi:noNamespaceSchemaLocation="http://intern.sig.nl/software-
            improvers/schemas/scope.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <scope srcDir="/tmp/test1" baseDir="/tmp">
        <modules/>
        <language name="C">
            <source overrideDefaultFilters="false"/>
        </language>
        <language name="Cpp">
            <source overrideDefaultFilters="false"/>
        </language>
        <language name="Java">
            <source overrideDefaultFilters="false"/>
            <test overrideDefaultFilters="false"/>
        </language>
    </scope>
</system>
```

Figure 4.1: An example of a *scope* configuration file

Modules information: which modules compose the system, the name of each module, which files are included or excluded in each module (also using boundaries information).

For customizing which files are associated with each technology or module, and which files are exception to general rule, there are also elements to serve this purpose:

Boundaries: defines a list of inclusions and/or exclusions and exceptions in order to compound the desired set of files.

Exclusions: defines which files are excluded from the analysis process. An example of its use is for instance to svn information files, that stay aside the source code, but have no relevance to the system itself.

Although the creation of the configuration became easier with the *scope* configuration, it is still necessary to manually inspect the system files, checking which technologies compose the software system.

To assist this task, SIG had already developed a tool to inspect a software system folder and produce a textual report containing the total number of files, total lines of text. This report grouping also listed total number of files per extension and distribution of files per folder. An example of such a report is listed in Figure 4.2.

Listing 4.2: system_report.txt

```
Text: 17 files with 7 extensions
java = 8
mak = 2
h = 2
cxx = 2
list = 1
c = 1
'no extension' = 1

Total Lines: 7160
cxx = 5469
mak = 903
java = 496
h = 108
'no extension' = 97
c = 86
list = 1

Files per directory:
gtk
    mak = 2
    h = 2
    cxx = 2
    list = 1
    c = 1
    'no extension' = 1
scope
    java = 8

-------------------
Executables: 0 files with 0 extensions

-------------------
Binaries: 1 files with 1 extensions
dtp = 1

Files per directory:
gtk
    dtp = 1

-------------------
[The following files are not included in the above statistics]
Empty files
    gtk/main

Unclassified
    gtk/instructions.pdf [pdf document, version 1.4]

Unreadable (check permissions)
    /tmp/test1/gtk/preferences.ini
```

Figure 4.2: The old system overview report

4.1 Detection of Technologies

In order to avoid the manual work of checking which are the available technologies present in a set of files, and which of these should be associated with which technology it was necessary to find a method to detect which and distinguish which files belong to each technology. We found several alternatives to perform technology detection on a set of files:

Extension based detection: Perhaps the most easy way to associate files with a specific context, is through their file extension. The original purpose of file extensions is to categorise files by different categories.

Semantic analysis classification: Investigating the structure and semantics of the textual contents of files.

`File` **utility detection:** using the unix `File`[Web 19] utility.

Filter-based detection: Using composed filters per technology that define rules for the classification of a file towards each technology.

Machine Learning Techniques: Using algorithms such as for example artificial neural networks, Bayesian networks, Decision trees, support vector machines.

For the detection of technologies, we opted by Filter-based detection. We created default group of filters per technology, and so we could use these groups of filters to detect the relation of each file with the technologies that the SAT can analyse. These groups of filters can be composed of several filters. For instance filters that mimic the disjunction of conjunction of other rules or filters that check for the extension of the filename. This allows SIG employees to use simple logic and to composite rules to obtain the desired set of files associated with a certain context.

4.2 Generated source code and pattern-based files classification

Some software systems are also composed by files that are created by source code generator tools. Source code generation is the act of generating source code basing on an ontological model such as a template and is accomplished with programming tools such as a template processor or an Integrated Development Environment (IDE). It is the SOP to separate files developed by the client's development team from source code generated files, in order not to obfuscate the data derived from the files, grouping their metrics separately.

To perform this classification task, SIG analysts normally perform manual inspection of files, relying on tools such as `grep`, `find` and `bash` scripts in order to locate source code generated files. To automate this task and make it easier to find such cases, we checked for patterns using regular expressions in the contents of each file. Here's a list of the some of regular expressions we used for identifying generated source code in the contents of the files:

Listing 4.3: GeneratedCodeReport.txt

```
<!-- *** Context for unique occurrence -->
<!-- Regular Expression: '((machine|automatically)[ ]+generated)' -->
<!-- Match: '/* DO NOT EDIT - FILE AUTOMATICALLY GENERATED FROM THE .syn FILE *' -->
<!-- Line: 2 -->
  <exclude pattern=".*/shader/arbprogram_syn.h"/>
  <exclude pattern=".*/shader/slang/library/slang_pp_directives_syn.h"/>
  <exclude pattern=".*/shader/slang/library/slang_pp_expression_syn.h"/>
  <exclude pattern=".*/shader/slang/library/slang_pp_version_syn.h"/>
  <exclude pattern=".*/shader/slang/library/slang_shader_syn.h"/>

<!-- *** Context for unique occurrence -->
<!-- Regular Expression: '(generated[ ]+(by|using|automatically))' -->
<!-- Match: '/* DO NOT EDIT - This file generated automatically by gl_table.py (from Mesa) script *' -->
<!-- Line: 1 -->
  <exclude pattern=".*/mesa-7.2/src/mesa/glapi/dispatch.h"/>
  <exclude pattern=".*/mesa-7.2/src/mesa/glapi/glapitable.h"/>
  <exclude pattern=".*/xorg-server-1.5.3/dispatch.h"/>
  <exclude pattern=".*/xorg-server-1.5.3/glapitable.h"/>
  <exclude pattern=".*/xorg-server-1.6.0/dispatch.h"/>
  <exclude pattern=".*/xorg-server-1.6.0/glapitable.h"/>

<!-- *** Context for several occurrences -->
<!-- Regular Expression: '(generated[ ]+(by|using|automatically))' -->
<!-- Match: 'printf("/* Automatically generated by vtable_layout_x86.cpp */\n")' -->
<!-- Line: 57 -->
<!-- Regular Expression: '(generated[ ]+(by|using|automatically))' -->
<!-- Match: '// Try to determine the vtable layout generated by G+' -->
<!-- Line: 5 -->
  <exclude pattern=".*/src/libs/xpcom18a4/xpcom/reflect/xptcall/src/md/unix/vtable_layout_x86.cpp"/>

<!-- *** Context for several occurrences -->
<!-- Regular Expression: '\#line' -->
<!-- Match: '#line 56 "parser.y' -->
<!-- Line: 73 -->
<!-- Regular Expression: '\#line' -->
<!-- Match: '#line 92 "parser.h' -->
<!-- Line: 91 -->
  <exclude pattern=".*/x11/x11include/xorg-server-1.5.3/parser.h"/>
  <exclude pattern=".*/x11/x11include/xorg-server-1.6.0/parser.h"/>
```

Figure 4.3: The generated code report

- `"(generated[ ]+(by|using|automatically))"`
- `"((machine|automatically)[ ]+generated)"`
- `"// Generated stubs for lazy dynamic linking"`
- `"// Generated headers for lazy dynamic linking"`
- `"// Generated Header File.  Do not edit by hand."`
- `"// Microsoft Developer Studio generated include file."`
- `"Unicode mapping table generated from"`
- `"generated from the DocBook documentation."`
- `"\\#line"`

This detection is quite error-prone. It ensures that most of source code generated files are found, but also false positives – files that are manually created but are matched by at least one from the above regular regular expressions.

To report the found occurrences/matches, we grouped files by their characteristics concerning which regular expression was used, what was the matched expression and in which absolute location in each file (line number). This is shown in Figure 4.3. This allowed analysts to easily verify the list of possible generated source code files, identifying groups of files that are most probably generated in the same way. Equally, it allows fast detection of probable false positives.

4.3 Scope Definition Generation and Findings Reports

In Figure 4.4 it is shown the improved system overview report that we generate with our tool. We report found technologies, its related subsets and a summary of the number of files per extension associated with each subset.

The tool performs a general classification that follows the algorithm shown in Listing 4.5. It first detects which files in a software system folder are source code files. For these files then checks for generated code and other exclusion patterns. If a file is associated with more than one technology, it reports it as an ambiguous file.

To generate the scope configuration file, the tool first builds the data objects and then recurs to the JAXB RI API to marshall these, generating the final XML representation of the configuration.

4.4 Scope validation

After a first generation, the scope configuration is usually refined by SIG employees. Because of this fact, we created an option in the tool so that it can be used to validate a refined scope. Instead of detecting which files are associated with the available technologies at the SAT, it simply performs the general classification in order to produce the new system overview report. This way, a SIG analyst can validate the scope configuration of a software system without having to perform the analyses. He refines and validates the scope configuration until it is as expected.

Listing 4.4: ScopeGenerationReport.txt

```
System Analysis Toolkit v. 1.41
Copyright Software Improvement Group
-------------------------------
  Found Technologies:
-------------------------------
C:
 contexts:
   source: 5
 extensions:
   h:2 cxx:2 c:1
Cpp:
 contexts:
   source: 5
 extensions:
   h:2 cxx:2 c:1
Java:
 contexts:
   source: 8
 extensions:
   java:8

-------------------------------
  Files Categorization Statistics:
-------------------------------
TOTAL Files: 32
   java:8 svn-base:8 (without extension):5 mak:2
   h:2 cxx:2 list:1 dtp:1 c:1 pdf:1 ini:1

Matched Files (Source and Test contexts): 8
   java:8
Ambiguous Files: 5
   h:2 cxx:2 c:1
Unmatched Files: 19
   svn-base:8 (without extension):5 mak:2 list:1
   dtp:1 pdf:1 ini:1
Detected Possible Trash Files: 0
Scope Excluded Files (by Pattern): 0
Generated Code Files (Generated context): 0
Generated Code Files (global patterns detection): 0

-------------------------------
  Generated Code:
-------------------------------
```

Figure 4.4: The new Scope Generation Report: an improved version of the System Overview Report

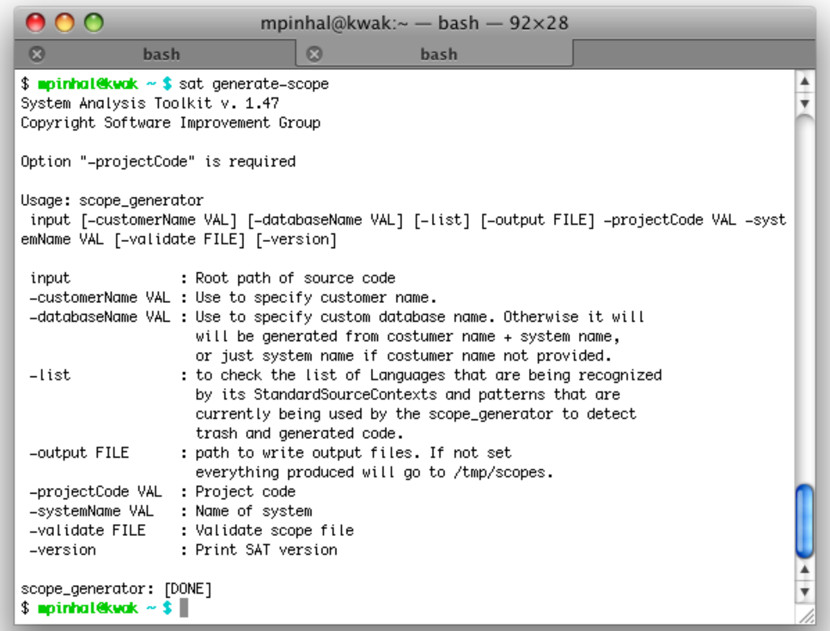
```
Input: Files
Output: FileInfo
foreach file do
    if is recognized by any technology detector then
        if is recognized by more than one technology detector then
            FileInfo ← AMBIGUOUS;
        else
            if recognized by generated code technology detector then
                FileInfo ← GENERATED_CODE_SUBTYPE;
            else
                FileInfo ← MATCHED;
        if is recognized by any trash detector then
            FileInfo ← TRASH;
        if is recognized as generated code then
            FileInfo ← GENERATED_CODE;
    else
        FileInfo ← UNMATCHED;
end
```

Figure 4.5: Scope Generator classification algorithm

4.5 Conclusions

With the created tool is now possible to generate a scope configuration in a fully automated way. SIG analysts don't need to create a new configuration based in an older one, and can now validate a scope configuration without having to run the analyses on a software system. The introduced tool saves time to the analysts and further streamlines the workflow process of software analyses. SIG analysts still need to validate the results, but have already a formed valid configuration. The tool also provides analysts with a clearer overview of a system through its reports and its automatically generated scope configuration.

The created tool (Figure 4.6) is already in use at SIG it is used in more than 90% of the cases.



```
$ mpinhal@kwak ~ $ sat generate-scope
System Analysis Toolkit v. 1.47
Copyright Software Improvement Group

Option "-projectCode" is required

Usage: scope_generator
 input [-customerName VAL] [-databaseName VAL] [-list] [-output FILE] -projectCode VAL -syst
emName VAL [-validate FILE] [-version]

 input             : Root path of source code
 -customerName VAL : Use to specify customer name.
 -databaseName VAL : Use to specify custom database name. Otherwise it will
                     will be generated from costumer name + system name,
                     or just system name if costumer name not provided.
 -list             : to check the list of Languages that are being recognized
                     by its StandardSourceContexts and patterns that are
                     currently being used by the scope_generator to detect
                     trash and generated code.
 -output FILE      : path to write output files. If not set
                     everything produced will go to /tmp/scopes.
 -projectCode VAL  : Project code
 -systemName VAL   : Name of system
 -validate FILE    : Validate scope file
 -version          : Print SAT version

scope_generator: [DONE]
$ mpinhal@kwak ~ $ 
```

Figure 4.6: Scope Generator tool usage help screen

Chapter 5

Graphical User Interface

In this chapter we present the development of the GUI of our integrated environment. We decided to focus the development of the GUI in the scoping task, since its the only part of the process that requires manual intervention and the use of external tools and also due to the lack of available time. Although we defined concrete objectives for our integrated environment and GUI, these objectives can be achieved through the combination of different sorts of interactions and UI elements. Because of this, we perform Interaction Design in order to obtain a design that achieves the expected usability and experience of our GUI.

To select the technology to build the GUI of the SAT, we decided to perform a comparative research study, defining a criteria that takes into account the built UI design and the previously elicited general objectives and requirements for our integrated environment.

We conclude by briefly documenting the implementation of the GUI in the selected technology, presenting the architecture of our application. We end by presenting the final result achieved.

5.1 Interaction design

Interaction design defines the behavior (the "interaction") of an artifact or system in response to its users. In Figure 5.1 we show the different resources used in our interaction design. We used different methods to obtain feedback from the future users of our tool.

We started by using pen and paper to gather the first generic ideas about how the interaction should occur. After the first ideas surface, we mockup these into images composed of UI elements, simulating the appearance of the UI. In order to simulate the behaviour of the UI, we used Microsoft Powerpoint to simulate the different interactions with the tool and the changes caused in the UI.

When consensus is achieved, we selected the technology to build our GUI, and we start implementing the individual components that will compose the final product. Finally we put them together in place and connect them in order to achieved the desired interaction, creating a working version of our GUI.

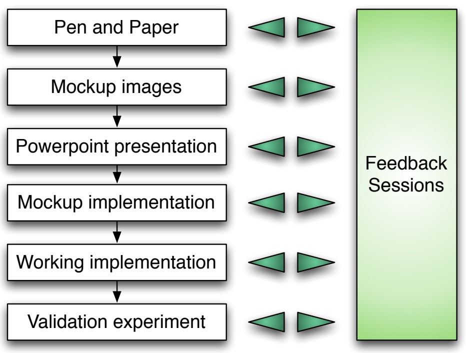


Figure 5.1: Interaction Design stages

5.2 Technology Selection

5.2.1 Criteria

To find and compare technologies, we define the following criteria.

Integration with SIG's tools How well and how easy is the integration with the SAT, specifically with the current technologies that compose it and with its development methodology, testing and release.

Available Libraries, technology support and documentation

UI Possibilities The availability of a good set of basic GUI elements: buttons, text fields, positioning and layout capabilities, ready-to-use *widgets* and more complex elements such as *drop-boxes*, *tree-lists*, etc.

Maintainability If there is expertise and knowledge about the technology in SIG and the source code maintainability of the technology. Also, if there is good separation of concerns, with the separation of the business layer and the presentation layer.

Testability The support of unit testing, the availability of frameworks to support it.

Debugging Which possibilities of debugging exist and how reliable it is, i.e., how verbose it is, performance penalty, etc.

Toolkit Support Which tools support the technology, which IDEs support it and to what extent the tool supports the technology.

Community Support The existence of mailing lists, forums, the size of the community and its latest activity.

Maturity For how long does the technology exist, how it has evolved and its current stability and maintainability state.

License and price If the license allows SIG to use the tool, if the license is paid, how affordable it is, etc.

5.2.2 Technologies

These was the final set of technologies that were subject of our comparative study:

Adobe Flex[Web 11] Adobe Flex is a software development kit for the development and deployment of cross-platform rich internet applications based on the Adobe Flash platform.

Java Swing[Web 09] Swing is a widget toolkit for Java. It is part of Sun Microsystems' Java Foundation Classes (JFC) - an Application Programming Interface (API) for providing a GUI for Java programs.

JavaFX[Web 10] JavaFX is a software platform for creating and delivering rich Internet applications that can run across a wide variety of connected devices.

Google Web Toolkit[Web 08] Google Web Toolkit is an open source set of tools that allows developers to create and maintain complex Javascript front-end web applications in Java.

Struts[Web 07] Apache Struts is a open-source framework for creating Java web applications. It uses and extends the Java Servlet API to encourage developers to adopt a Model-View-Controller (MVC) architecture. Its main design goal is to cleanly separate the model (application logic that interacts with a database) from the view (HTML pages presented to the client) and the controller (instance that passes information between view and model).

Apache Wicket framework[Web 03] "Wicket bridges the mismatch between the web's stateless protocol and Java's OO model. The Component Based Wicket Frameworks shields you from the HTTP under a web app so you can concentrate on business problems instad of the plumbing code. In Wicket you use logic-free Hyper Text Markup Language (HTML) templates for layout and standard Java for an application's behaviour"[5].

5.2.3 Findings

Table 5.1 summarises our comparison between the different technologies. All technologies accomplish the criteria mentioned above. Among this criteria, the ones we gave more importance were the integration with the SAT (Java, Maven2[Web 12]), the maintenance and testability, the toolkit support (plugin for Eclipse is ideal) and the UI possibilities.

From the set of researched technologies, Swing is the most mature. Struts has been evolving for some years now, having merged from two precedent frameworks (Struts[Web 06] and WebWork[Web 05]). The remainder technologies have just emerged in the past couple of years.

About toolkit support, the development in all technologies can be performed through Eclipse with support of edition, compilation, testing and debugging. Flex

	Flex	GWT	JavaFX	Swing	Struts2	Wicket
Integration	o	++	++	++	+	++
Libraries	+	++	+	++	++	++
UI elements	+	+	+	+	+	++
Maintainability	o	o	-	+	o	+
Testability	+	+	o	o	+	++
Toolkit Support	++	+	o	+	+	+
Debugging	+	+	++	++	o	++
Community Support	++	+	-	-	+	++
Maturity	+	o	-	++	+	+
License and price	o	++	++	++	++	++
Experimentation	o	+	+	+	o	+
Overall	+	+	o	+	o	++

Table 5.1: Researched Technologies Comparison.
The scale is the following: −,-,o,+,++

and JavaFX require their own plugins for Eclipse. The others rely on common plugins for Java SE/EE such as the Java Development Tools (JDT) plugin[Web 13]. About the Testability criteria, all of them offer frameworks for the purpose of Unit Testing, but some more comprehensive than others. From this criteria, Wicket is the technology that offers most flexibility, being also possible to perform integration tests mimicking UI events and state evolution through the `WicketTester` classes in conjugation with the JUnit Testing Framework[Web 14].

All technologies were free to use by SIG, and provided good IDEs for the development of the application. Just Adobe Flex would require the purchase of a license of the Adobe Flex Builder Pro as plugin for Eclipse or NetBeans. All the others with the exception of Swing develop and maintain their own plugins for Eclipse.

After analysing the results of our investigation, we were mainly in doubt for GWT and Apache Wicket. Maintenance was the most discriminating factor. Flex's integration with SIG's SAT would be the most complex, The poor toolkit support in Eclipse of Swing, the difficult integration with Adobe Flex (using for instance BlazeDS[Web 15] or the LiveCycle Enterprise Suite[Web 16] as for Java remoting and web messaging technology), the poor separation of concerns in JavaFX and and Struts2 heavy MVC architecture were the main reasons for exclusion. As final discriminating factor, we excluded GWT due to the lack of maturity, taking in consideration the uncertain path of Google's project. We found the most suitable technology for the development of our GUI to be the Apache Wicket framework. The concise community support and resources, the good separation of concerns, the offered UI possibilities and the testing capabilities seemed to meet our main requirements.

5.3 Implementation

5.3.1 Wicket Way - `Pages`, `Panels` and `Components`

All that concern the management of state, sessions with the users, and HTML requests, is handled transparently by Wicket. This allows developers to concentrate on solving business problems rather than writing plumbing code.

Conceptually, the `Application` object is the top-level container that bundles all components, markup and properties files, and configuration. With Wicket we program our `component`s and `page`s using regular Java constructs. You create components using the `new` keyword, create hierarchies by adding child `component`s to parents, and use the `extend` keyword to inherit the functionality of other `component`s. The presentation part the web application is defined in HTML templates. Any Wicket component that requires view markup in the form of HTML needs to be side-by-side with the Java file.

5.3.2 Architecture

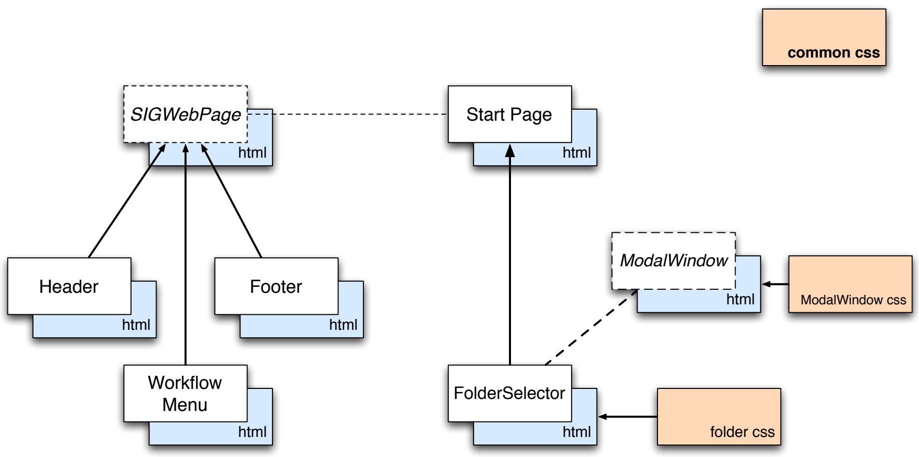


Figure 5.2: GUI elements hierarchy with Wicket

Our architecture is simply defined by an abstract skeleton, a set of pages that implement our desired hierarchy of components. Figure 5.2 presents the main components that form the basic architecture of our application. We do not include the predefined `Application`, that just points to `StartPage`, as well our extended `Session` that keeps track of the data inputted by the user.

The abstract class `SIGWebPage` holds itself several `Panel`s forming a skeleton/template for any other class that implements it.

To achieve easy customisation and reuse of components, we started by defining an Hierarchy for components. Also in Figure 5.2, we present an example of such customization: `StartPage` implements the skeleton class `SIGWebpage`, and through this, it will be composed of the `Workflow Menu`, `Header` and `Footer`.

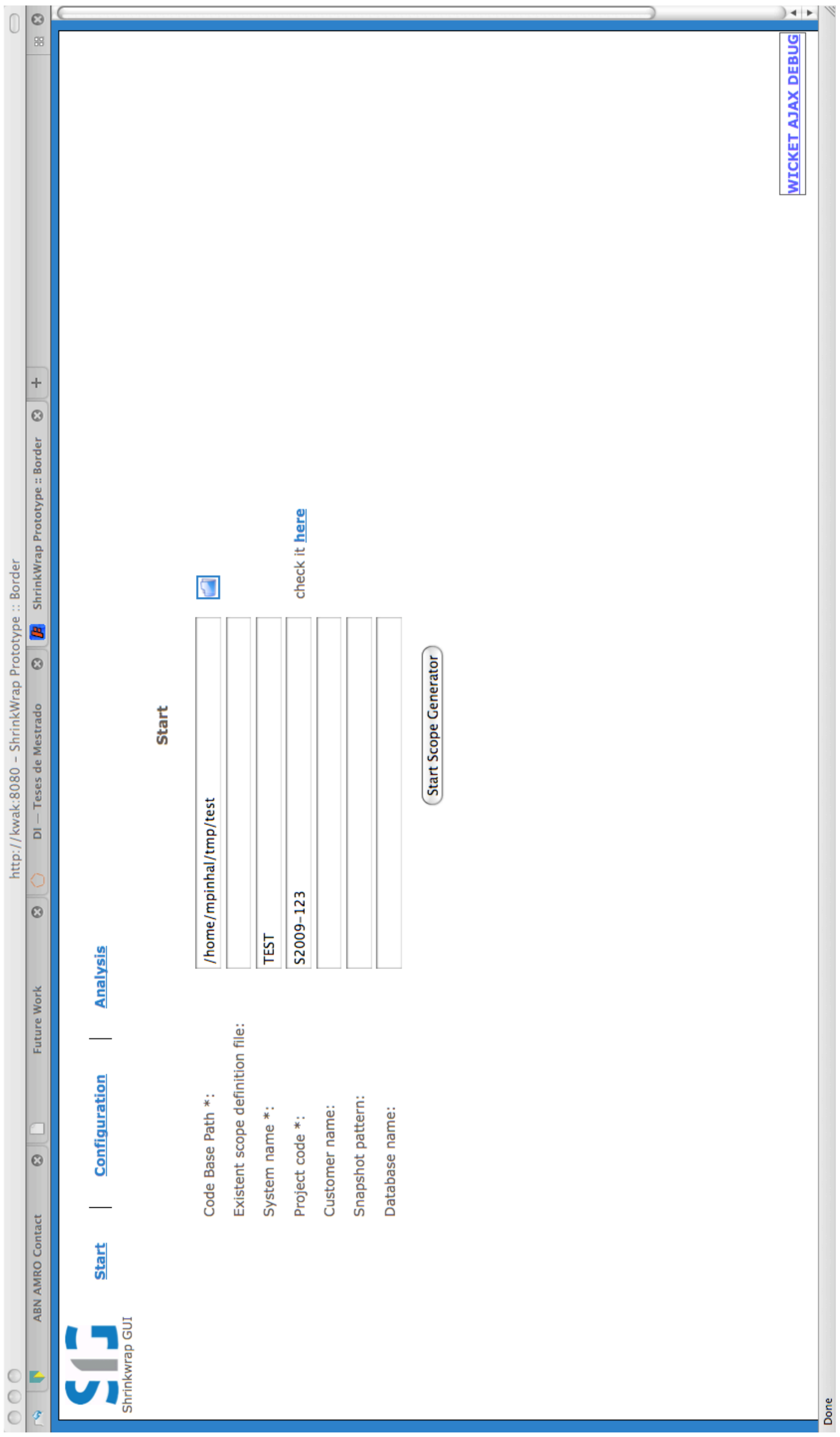
5.3.3 Tooling and Setup

For the development our GUI we used the Eclipse IDE[Web 02] (version 3.3.2), with the usual plugins that support the development of Java Systems. For the deployment and dependencies management we used Maven 2[Web 12]. For debugging we used another Eclipse Java debugging capabilities with the Run Jetty Run[Web 22] plugin, that more than just allowing debugging, it allows hot deploy, the ability of updating some parts of the implementation of java methods or classes, as long as the structure is not changed.

5.4 Result

In Figures 5.3 to 5.6 we present some screenshots of our developed GUI application. In Figure 5.4 we show how the interface displays which files were matched by technologies detectors, which files weren't and which files were excluded from the analysis. On the right there is a file browser that allows users to visualise the files associated with a context from the left, by filtering by extension and folder. This allows the user to personalise the list of files and to easily inspect the files (Figure 5.5).

Furthermore the user can refine the configuration at different levels. He can assign extensions, folders or files to a certain technology, or discard them as intended. This is performed by selecting the drop-boxes available at each level of the file browser. The scope configuration is then refined and the user does not need to edit the XML scope configuration file, being able to run the analyses when he completes the scope configuration.

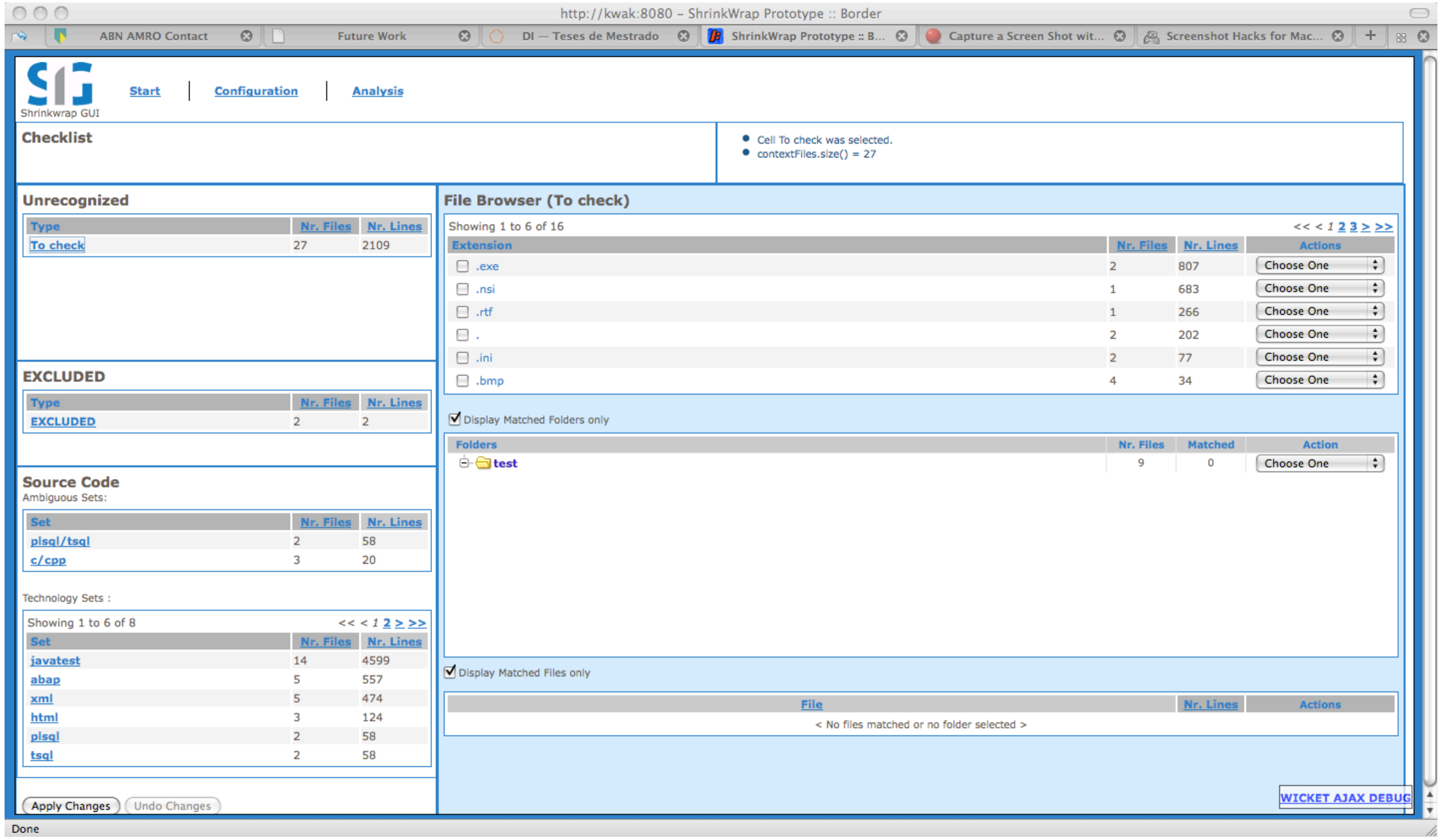


Start

Code Base Path *:	/home/mpinhal/tmp/test
Existent scope definition file:	
System name *:	TEST
Project code *:	S2009-123
Customer name:	
Snapshot pattern:	
Database name:	

check it here

Figure 5.3: GUI - Start screen


http://kwak:8080 – ShrinkWrap Prototype :: Border
ABN AMRO Contact
Future Work
DI — Teses de Mestrado
ShrinkWrap Prototype :: B...
Capture a Screen Shot wit...
Screenshot Hacks for Mac...
Shrinkwrap GUI
Start | Configuration | Analysis

Checklist

- Cell To check was selected.
- contextFiles.size() = 27

Unrecognized

Type	Nr. Files	Nr. Lines
To check	27	2109

EXCLUDED

Type	Nr. Files	Nr. Lines
EXCLUDED	2	2

Source Code
Ambiguous Sets:

Set	Nr. Files	Nr. Lines
plsql/tsql	2	58
c/cpp	3	20

Technology Sets :

Showing 1 to 6 of 8 << < 1 2 > >>

Set	Nr. Files	Nr. Lines
javatest	14	4599
abap	5	557
xml	5	474
html	3	124
plsql	2	58
tsql	2	58

File Browser (To check)

Showing 1 to 6 of 16 << < 1 2 3 > >>

Extension	Nr. Files	Nr. Lines	Actions
.exe	2	807	Choose One
.nsi	1	683	Choose One
.rtf	1	266	Choose One
.	2	202	Choose One
.ini	2	77	Choose One
.bmp	4	34	Choose One

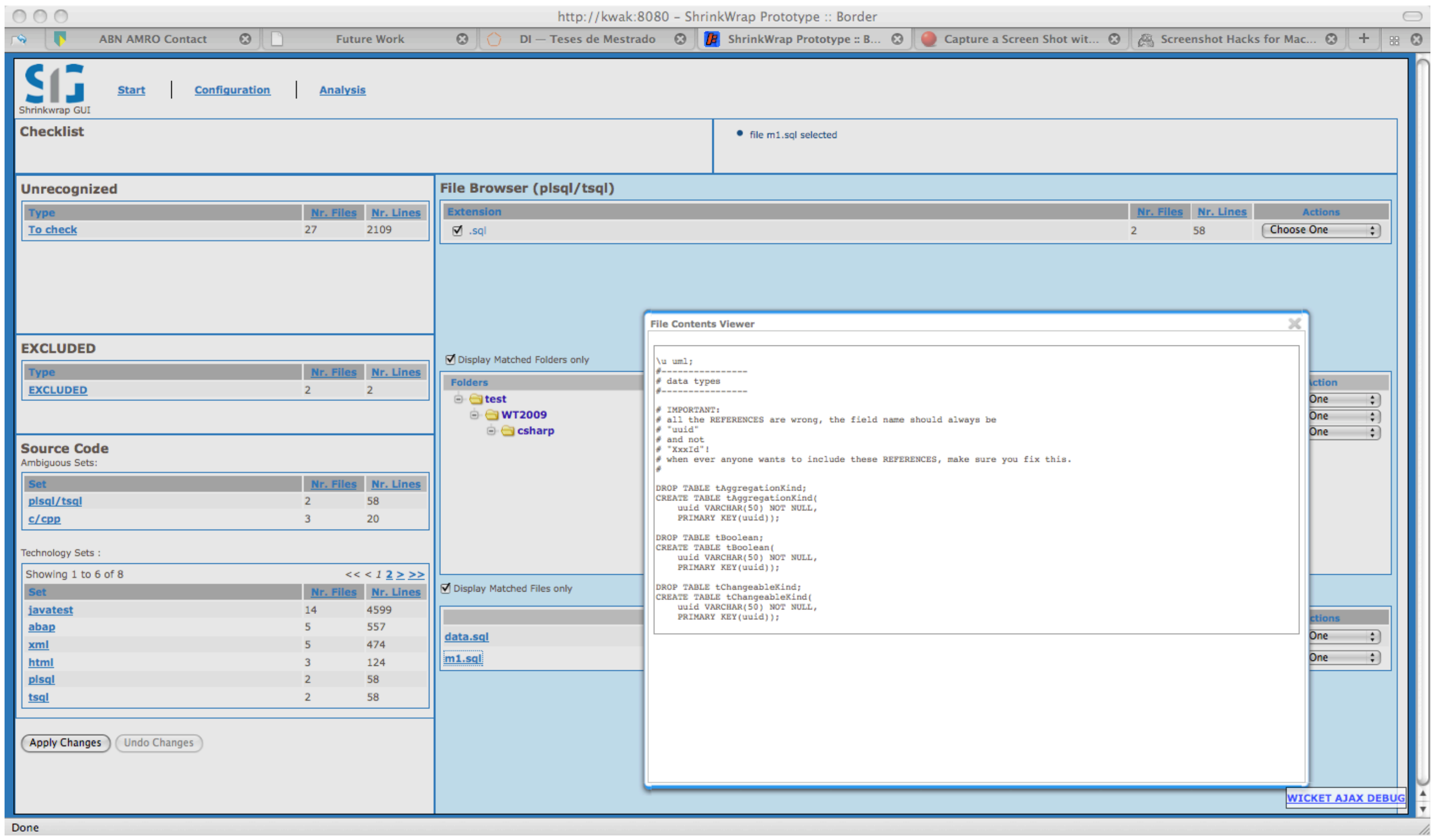
✓ Display Matched Folders only

Folders	Nr. Files	Matched	Action
test	9	0	Choose One

✓ Display Matched Files only

File	Nr. Lines	Actions
< No files matched or no folder selected >		

Figure 5.4: GUI - Scope Configuration screen



Type	Nr. Files	Nr. Lines
To check	27	2109

Type	Nr. Files	Nr. Lines
EXCLUDED	2	2

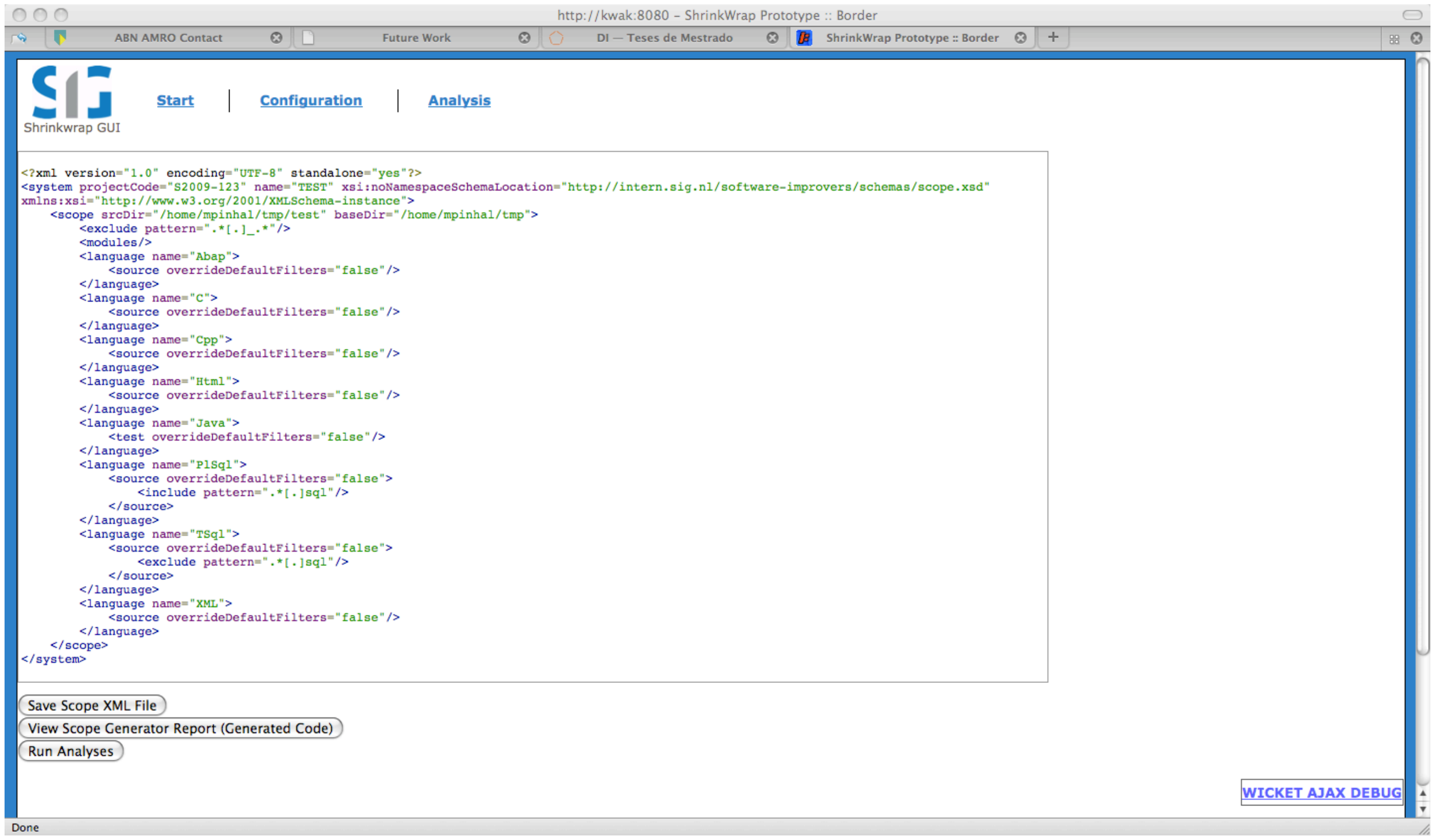
Set	Nr. Files	Nr. Lines
plsql/tsql	2	58
c/cpp	3	20

Set	Nr. Files	Nr. Lines
javatest	14	4599
abap	5	557
xml	5	474
html	3	124
plsql	2	58
tsql	2	58

Extension	Nr. Files	Nr. Lines	Actions
✓ .sql	2	58	Choose One

```
\u uml;
#----------------
# data types
#----------------

# IMPORTANT:
# all the REFERENCES are wrong, the field name should always be
# "uuid"
# and not
# "XxxId"!
# when ever anyone wants to include these REFERENCES, make sure you fix this.
#

DROP TABLE tAggregationKind;
CREATE TABLE tAggregationKind(
    uuid VARCHAR(50) NOT NULL,
    PRIMARY KEY(uuid));

DROP TABLE tBoolean;
CREATE TABLE tBoolean(
    uuid VARCHAR(50) NOT NULL,
    PRIMARY KEY(uuid));

DROP TABLE tChangeableKind;
CREATE TABLE tChangeableKind(
    uuid VARCHAR(50) NOT NULL,
    PRIMARY KEY(uuid));
```

Figure 5.5: GUI - Scope Configuration screen - inspecting a file



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<system projectCode="S2009-123" name="TEST" xsi:noNamespaceSchemaLocation="http://intern.sig.nl/software-improvers/schemas/scope.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <scope srcDir="/home/mpinhal/tmp/test" baseDir="/home/mpinhal/tmp">
        <exclude pattern=".*[.]_.*"/>
        <modules/>
        <language name="Abap">
            <source overrideDefaultFilters="false"/>
        </language>
        <language name="C">
            <source overrideDefaultFilters="false"/>
        </language>
        <language name="Cpp">
            <source overrideDefaultFilters="false"/>
        </language>
        <language name="Html">
            <source overrideDefaultFilters="false"/>
        </language>
        <language name="Java">
            <test overrideDefaultFilters="false"/>
        </language>
        <language name="PlSql">
            <source overrideDefaultFilters="false">
                <include pattern=".*[.]sql"/>
            </source>
        </language>
        <language name="TSql">
            <source overrideDefaultFilters="false">
                <exclude pattern=".*[.]sql"/>
            </source>
        </language>
        <language name="XML">
            <source overrideDefaultFilters="false"/>
        </language>
    </scope>
</system>
```

Figure 5.6: GUI - Analysis screen

Chapter 6

Experimental Validation

To evaluate the created GUI, we decided to perform a validation experiment. The purpose of this experiment is a first quantitative evaluation of the usefulness of the created integrated environment for the purpose of creating a scope configuration. Similar to [2] we assess the usefulness of the new tools, maintaining a distinction between time spent and correctness.

Although our integrated environment addresses other aspects of the general software analysis procedure, by standardising and improving its consistency, we focus the experiment in the analysts task of creating a scope configuration. This is due to the fact that its the only part of the process that requires manual intervention and the alternative use of external tools, that can actually be assessed and measured in terms of time and correctness.

6.1 Experimental Design

In this section we present a detailed description of the design of our experiment.

Research Questions

- *Is it faster to create a scope configuration with the new set of tools in comparison to the previous set of tools?*
- *Does the new Integrated Assessment contributes for the correctness of configurations?*

To answer this questions, we shall measure time and analyse the scope configurations produced by our subjects for correctness.

Objects We decided to use two systems as objects of our experiences. In Table 6.1 are shown their details. For anonymity reasons we use anonymous names. The criteria for the selection of these two systems, were the following:

- They are representative of typical software systems to be analysed by SIG employees. Both systems contain an average amount of technologies, and contain generated code and configuration files.
- Both systems were analysed before at SIG, so their scope configuration was done earlier by another analyst, and approved by the client.
- The systems are unknown to the subjects of this experience.
- The systems contain the same key characteristics: same key technology, total number of lines of text representative of usual systems analysed at SIG, contain source code generated files and have about the some amount of number of extensions in the whole set of files.

	System X	System Y
main technology	Java	Java
other technologies	JSP, Javascript	Html, XML
lines of code	35.000	100.000
# files included	320	1500
# total files	1250	1800

Table 6.1: The two systems objects of our experiment

Task design With respect to the tasks to be tackled during the experiment, we maintain two important criteria:

- It should be representative of real analysis contexts at SIG.
- It should be a well known task for the subjects.
- It should not be biased towards our tool or the traditional procedure.

The task is to create a configuration as the SOP describes. More details in this procedure can be found in Chapter 4.

Subjects The subjects in this experiment were 4 SIG analysts. All of them participate on a voluntary basis and can therefore be assumed to be properly motivated. All of them have prior experience in creating scope configurations, performing this task on a regular basis. We consider the letters A, B, C and D to represent each subject during our experiment.

Experimental procedure We created two groups, “AB” and “CD”, the first composed by subjects A and B and the latter by subjects C and D. Subjects perform the task individually. The experiment is done in 2 rounds. In the first round, each subject from group AB performs the task with the support of the GUI tool, while the subjects from group CD perform the task with the old set of tools. In the second round, the positions are inverted and the group that performed before with the support of the GUI tool, performs now with the old set of tools and vice versa. The sessions are conducted on workstations with equal characteristics, i.e., all of them with the same characteristics in what concerns:

processing power, amount of RAM and screen resolution. At the beginning of each round each subject is given a sheet (1 page) with detailed instructions, and for the group that performs the task with the support of the GUI tool, there is also a small demo of how the tool works. Subjects have a limit of 45 minutes, which is the average time necessary to perform a scope configuration. To gather feedback about our tool, we wrote a questionnaire to be answered in the end of the procedure. The questionnaire contains questions about its functionality and usability, where the user is invited to rate these in common rating scales of Agreement and Importance[Web 17].

6.2 Results

The results of the two rounds are presented at Tables 6.2 and 6.3.

Setup	Subject	Valid XML	Technologies	Generated Code	Time
With Tool	A	yes	yes	yes	**32 m.**
	B	yes	yes	yes	45 m.
Without tool	C	yes	yes	**no**	45 m.
	D	yes	yes	yes	45 m.

Table 6.2: Results of 1st round: System X

Setup	Subject	Valid XML	Technologies	Generated Code	Time
With Tool	C	yes	yes	yes	45 m.
	D	yes	yes	yes	**38 m.**
Without tool	A	yes	**no** (2/3)	yes	**40 m.**
	B	yes	yes	yes	45 m.

Table 6.3: Results of 2nd round: System Y

Time results All the subjects were able to perform the task in time. Most of subjects needed the total available time with the GUI tool and without it, with the exception of subject A, that performed both tasks in shorter amount of time. Both subjects A and D performed the task with the support of the tool in less amount of time.

Correctness results All the subjects performed correct scope configurations with the support of the tool. Subject C did not find the generated source code at round 1, and Subject A did not include a technology of the system to the scope configuration.

6.3 Discussion

From the analysis of the results, we verify that the experience revealed that our built GUI answers both research questions positively. Although we don't have significant statistical data to validate the results, two of the four subjects performed faster with the support of the new set of tools in comparison to the previous set of tools, and the remainder two took the same time. Regarding correctness, two of the subjects obtained an incorrect scope configuration when performing the task without the new set of tools, and all of the subjects obtained a correct scope configuration using the new set of tools.

6.3.1 Threats to validity

After the experiment was done, analysing the design and the notes we took meanwhile the experience was performed, we recognise the following threats to validity:

Atmosphere and Interference One aspect that was not considered in our design, was the atmosphere around the subjects while these performed the experience. The subjects performed the experience at their usual working place, and we observed that there was noise and discussions involving other SIG analysts and consultants going at the same room where the subjects were performing the experiment. Some of the subjects were even interrupted several times, and this might have affected their concentration and performance.

Experience with the new tool Although a demonstration of the tool was given before the subjects performed the experiment, most of the subjects had difficulty on understanding some interaction details of the GUI. Because we introduced a new paradigm on performing a configuration, we should have considered that it is also important that the users are familiar with the tool. This might have lead to a significant increase of performance, and so although the results answer positively to our research questions, we could have achieved more significant numbers.

Statistical Significance As mentioned above, the statistical significance of this validation experiment is reduced, due to the small number of subjects. Despite this fact, the number of subject is a considerable percentage of SIG's analysts (around 30%).

Chapter 7

Conclusions

We successfully built an Integrated Environment for Software Assessments. We used several techniques in order to elicit the required information so that we could understand the services offered by SIG, the followed Standard Operating Procedures (SOP) and the tools that support the source code analysis performed by SIG. Then, we defined the necessary requirements for the development of our integrated environment in order to address the challenges reported by SIG's employees.

Through the path of standardisation, automation and integration we were able to achieve consistency, and also to avoid rework and prevent errors related to the source code analysis performed at SIG. More specifically, we standardised the installation of tools and its usage, we automated the configuration procedure for the analysis of software systems, and we integrated the diverse tools of the SAT in our developed IE. This brought consistency to the software analysis procedure, diminishing the related amount of rework and errors, supporting the assessments conducted by SIG.

In order to automate the full source code analysis procedure, we concentrated our efforts in its most relevant bottleneck: the definition of the scope configuration, that defines which specific analyses will be performed over which files. For this purpose, we created a tool that inspects the files of a software system, classifying these and generating a configuration in an automated manner. Instead of SIG analysts having to inspect a software system and having to create a configuration from scratch, the tool generates the most probable configuration based on solid assumptions and analysts need only to validate it and refine it only if necessary.

For the file classification we used filter based classification, having filters defined for the recognition of each technology. This allowed to easily define multiple criteria for inclusion or exclusion of files in a single composed filter. Still in the context of the tool, we also included the detection of generated source code files, that was one task that SIG analysts would have to perform recurring to other common command line tools and scripts. To further optimise this subtask of the configuration procedure the tool also generates a report listing possible generated source code files grouped by the same characteristics (used regular expression, matched expression, absolute location), facilitating the processing and categorisation of such files. This tool was successfully integrated into the SOP of software analysis. The procedure is now executed with fewer manual work and errors, standardising and automating the whole source code analysis procedure.

To standardise the use of the SAT, we integrated the several tools that compose it by developing a GUI. We performed a comparative research study and we selected the Apache Wicket framework for the development of our GUI. The GUI interface tightly tied the SAT with the SOP, ensuring the correct use of tools by providing a consistent workflow to their users.

With the availability of UI elements we decided to further optimise the configuration procedure. We performed interaction design and we built an interactive UI to the configuration tool previously created. In a short amount of time we were able to create a working UI that allowed SIG analysts to inspect a system from different points of view, in a dynamic but standard way.

We also performed a small validation experiment in order to evaluate the build tool in the task of configuring the scope of the analyses. The results achieved showed that the new IE can contribute for saving time and for making the analysts tasks easier.

7.1 Future Work

The scope configuration tool should be improved with a second round of recognition in order to detect ambiguous files – files that are currently associated with more than one technology by the tool. This is due to the simple detection method used by SIG, that relies on the file extension for the association. This second round of recognition could be performed using machine learning techniques, or other simpler techniques such as content keywords matching.

The current scope model, brought standardisation to the configuration procedure at SIG. The automatic generation of a scope configuration, in addition to the conciseness of its format, makes it easier and simpler for SIG analysts to validate it, and refine it as necessary. Despite this fact and although the new scope configuration format and procedure was successfully introduced in the standard procedure (is already in use by more than 90% of times), there are situations where an analyst still needs to manually create an old Spring configuration. In order to further standardise the procedure, it should be considered to extend the current scope model to replace the old configuration, as to avoid the maintenance of the two formats and related derived problems.

Further work can also be done in the GUI. By providing an interactive GUI, we were able to simplify the analysts task of performing a configuration and running the analyses for a given software system. Still, the GUI is still not completely reliable. The questionnaires collected in the validation experiment gave us feedback on what were the main problems that users, giving us good pointers for improvement. Concretely, related to the scope configuration GUI, the tool could also classify well known sets of non source code files, in order to facilitate the refinement and validation of the automatically generated scope. It also lacks the ability to validate the detected possible generated source code files, and the support for the definition of modules of a system. Using the developed browsing technique, the tool could allow the user to define the modules of the system interactively. Furthermore, the tool lacks the ability to allow analysts to inspect generated source code groups (see 4.2), providing the necessary interaction for discarding false positives.

The ability to dynamically compose and shuffle the order of the file browsing filter components could serve better the purpose of inspecting different kinds of systems. For instance, instead of browsing through a set of files by filtering extensions, folders and files (in this order), the analyst could opt to filter first by folder and then by extension. Further investigation on filtering components could reveal other filtering mechanisms over sets of files, that could further assist the inspection of a software system. Some components could be based on filtering mechanisms as for instance: file name patterns, repeated file names (could detect ambiguities in the received source code of a software system), or keyword detection (file contents).

One important aspect that should also be considered in the near future is the connection between the implemented GUI and all project resources and information about the performed analyses and generated deliverables (and IDs). This was not in the scope of our research, but such integration could ensure the automatic and correct storage of deliverables and analyses unique identifiers, further standardising the analysis process, and also diminish or even end the manual work that analysts follow in order to ensure the traceability of deliverables.

Acronyms

The Software Improvement Group related acronyms:

SIG	Software Improvement Group	3
SAT	Software Analysis Toolkit	v
SRA	Software Risk Assessment	1
SM	Software Monitor	1
SPM	Software Portfolio Monitor	7
SPC	Software Product Certification	1

Other acronyms:

API	Application Programming Interface	33
BPMN	Business Process Modelling Notation	5
CLI	Command line interface	12
HTML	Hyper Text Markup Language	33
IDE	Integrated Development Environment	24
IE	Integrated Environment	4
GUI	Graphical User Interface	i
MVC	Model-View-Controller	33
RIA	Rich Internet Application	17
SOP	Standard Operating Procedure	2
UI	User Interface	19
XML	eXtensible Markup Language	21

Bibliography

[1] Coiera, E. (2002). Interaction design.

[2] Cornelissen, B., Zaidman, A., Rompaey, B. V., and van Deursen, A. (2009). Trace visualization for program comprehension: A controlled experiment. pages 100–109.

[3] Correia, J. and Visser, J. (2008a). Benchmarking technical quality of software products. pages 297–300.

[4] Correia, J. and Visser, J. (2008b). Certification of technical quality of software products.

[5] Dashorst, M. and Hillenius, E. (2008). *Wicket in Action.*

[6] Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. *Quality of Information and Communications Technology, International Conference on the*, 0:30–39.

[7] Herrington, J. and Kim, E. (2008). Getting started with flex^{TM} 3. pages 1–148.

[8] ISO (2001). Iso/iec 9126-1: Software engineering - product quality - part 1: Quality model.

[9] Kuipers, T. and Visser, J. (2004). A tool-based methodology for software portfolio monitoring. pages 118–128.

[10] Mcphillips, T., Bowers, S., Zinn, D., and Ludäscher, B. (2009). Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541–551.

[11] Resources, A. L. (2008). Blazeds developer guide. pages 1–217.

[12] van Laer, T. (2008). Modelling, improving and executing a software risk assessment process.

[13] Vigder, M., Vinson, N. G., Singer, J., Stewart, D., and Mews, K. (2008). Supporting the everyday work of scientists: Automating scientific workflows. pages 1–15.

[14] Wasserman, A. (1990). Tool integration in software engineering environments. *Software Engineering Environments*, pages 137–149.

[15] Wicks, M. (2004). Tool integration in software engineering: The state of the art in 2004.

[16] Wicks, M. (2006). Tool integration within software engineering environments: An annotated bibliography. Technical report.

[17] Wicks, M. and Dewar, R. (2007). A new research agenda for tool integration. *Journal of Systems and Software*, 80(9):1569 – 1585.

Web References

[Web 01] The Software Improvement Group
http://www.sig.eu/

[Web 02] Eclipse Studio
http://www.eclipse.org/

[Web 03] Apache Wicket Framework
http://wicket.apache.org/

[Web 04] Wicket In Action - examples
http://code.google.com/p/wicketinaction/

[Web 05] WebWork Java web-application development framework
http://www.opensymphony.com/webwork/

[Web 06] Struts Java web-application development framework
http://struts.apache.org/1.x/

[Web 07] Struts2 Java web-application development framework
http://struts.apache.org/

[Web 08] Google Web Toolkit platform
http://code.google.com/webtoolkit/

[Web 09] Swing widget toolkit for Java
http://java.sun.com/javase/technologies/desktop/

[Web 10] JavaFX development platform
http://www.javafx.com

[Web 11] Adobe Flex is a software development kit
http://www.adobe.com/products/flex/

[Web 12] Apache Maven software project management and comprehension tool
http://maven.apache.org/

[Web 13] Java Development tools (JDT) Eclipse plugin
http://www.eclipse.org/jdt/

[Web 14] JUnit Java Testing Framework
http://www.junit.org/

[Web 15] BlazeDS server-based Java remoting and web messaging technology.
http://opensource.adobe.com/wiki/display/blazeds/BlazeDS/

[Web 16] Adobe LiveCycle Enterprise Suite 2
http://www.adobe.com/products/livecycle/

[Web 17] Common Rating Scales, Vovici
http://blog.vovici.com/blog/bid/18261/Common-Rating-Scales-to-Use-when-Writing-Questions

[Web 18] Spring application framework
http://www.springsource.org/

[Web 19] File utility *man* pages
http://linux.die.net/man/1/file

[Web 20] Java Architecture for XML Binding (JAXB)
http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

[Web 21] JAXB Reference Implementation
https://jaxb.dev.java.net/

[Web 22] Run-Jetty-Run Eclipse plugin
http://code.google.com/p/run-jetty-run/

[Web 23] Software Engineering Body of Knowledge (SWEBOK)
http://www.computer.org/portal/web/swebok