SWT tomorrow's software technologies

Universität Bamberg

Guidelines for the SWT-SWL-B Report

Language

The report shall be written in English or German.

Format

The report shall be printed on A4 paper, use a double-sided, single-spacing page format with reasonable margins (at least 15mm and at most 30mm to the left and right) and employ font *Computer Modern* or *Times* in size 12pt. All pages shall be numbered.

Structure & Content

The report's structure shall be the one of this document. In particular, the report shall contain a title page, a table of contents, a list of figures, all sections and subsections of this document, a bibliography, and an appendix with the final product backlog. Further appendices may be added as needed.

In the sequel, the expected content of each section is summarized in italics. It is strongly recommended that you use this document's LaTeX sources as a template for your group's report.

Expected Number of Pages

The report shall be 30–50 pages *of text* in length. This excludes the title page, the table of contents, the table of figures, the bibliography, all appendices and the Ehrenwörtliche Erklärung, as well as all figures, diagrams and code excerpts/listings.

Figures & Diagrams

Each figure, diagram or code excerpt/listing/table shall be easily readable and have a number and caption that also appears in the list of figures/tables. See Figure 1 and Table 1 as examples.

Figure 1: Example figure.

Table 1: Example table

Section number	1	2	3	4	5	6
Expected. no. of pages	2–3	6–12	5–8	10–15	4–7	3–5

References

Citations shall be marked in square brackets by an alphanumeric author-year system, e.g., [**?**, **?**] and [**?**]. Make sure that all sources are referenced properly and all bibliography entries are complete.

Ehrenwörtliche Erklärung

All group members shall sign the *Ehrenwörtliche Erklärung* (Declaration of Proper Academic Conduct) on the report's last page.

Please do not forget to justify in your report all technical and non-technical aspects of your group's conduct of the software development project.

Universität Bamberg

Report

SWT-SWL-B Software Engineering Lab

Winter Semester 2016/17

Group A

Frank Keßler	1742945	SoSySc/4
Andreas Köllner	17420191	AI/7?
Jan Martin	1796943	AI/5
Simon Meyer	1785554	WI/5
Tobias Schwartz	1738195	SoSySc/6

Supervisor: Prof. Dr. G. Lüttgen

Version: February 2, 2017

Contents

List of Figures

List of Tables

1. Project Organization

In this section we describe the overall goal of the project, the internal organisation and work distribution of our group, and our activities during the Blast-off of the project.

1.1. Goal of the Software

In this subsection we describe the goal of the Test Data Analyser (TDA). The TDA is developed for medatixx GmbH & Co. KG, a company that develops software for medical practices. TDA is an application to help software developers and testers at medatixx to analyse test data of their software. Purpose, advantage, and measurement are the three parts of which the goal of TDA is made of.

Purpose

During the development of their software, medatixx uses a sequence of builds, i.e. (pre-)release versions of the software. To ensure the quality of their product, each build is tested via a number of unit tests which are defined and executed on the classes of the corresponding build. The collection of these unit tests, their classes and the build they belong to, and their results is called a test run. TDA shall support the analysis of these test runs to help medatixx to discover builds with classes that are problematic to test.
To do so, TDA shall extract the necessary information from the test run XML files provided by medatixx, analyse the information in different ways including the usesage of the Apriori algorithm and visualise the results of the information analysis.

Advantage

The TDA provides new and more detailled information on the tests of different builds. It highlights the classes with the highest failure percentage of a specific test run. It shows the evolution of a class by visualising its failure percentages over multiple test runs. By using

the Apriori algorithm it shows possible associations between different classes. It offers an easy method to compare the tests of a specific class in different test runs.
With the additional information the TDA is making available for testers at medatixx, they get new insight into their testing methodology and the overall test quality is improved.

Measurement
Due to the easily accessible information on test runs and the discovered associations between classes, the resolution of failed classes and their corresponding unit tests will be accelerated by 50 %.

1.2. Organisation of the Group

Table 2: Distribution of work

Name	Responsibilities	Principal Artefacts	Work Time
Andreas	Identify user stories	User story cards	5
Andreas	Create stakeholder map	stakeholder map	3
Andreas	StAXParser	Methods for parsing	10
Andreas	Preparing paper prototype	Paper prototype	6
Andreas	Create high level architecture diagram	High level architechture diagram	4
Andreas	Documenting Sprint 2	Sprint 2 wiki	3
Andreas	Directory browser in GUI	Corresponding methods	3
Andreas	Tree view and handling for imported test runs	Corresponding methods	11
Andreas	Testing of classes in package logic without Parser, Analyzer	JUnit tests	12
Andreas	Create use case diagram	use case diagram	7
Andreas	Exception handling in StAXParser	Corresponding methods	2
Andreas	Documentation of Model	Javadoc in Model	2
Andreas	Documentation of Logic classes	Javadoc in Logic	8
Andreas	Author of chapter 1	Chapter in project report	10
Total Andreas			**total**

1.3. Project Blast-off

The Project Blast-off is the most important activity to decide whether or not to go ahead with a project. It is used to gather information on the project and make sure that it is viable and well founded.
Before we defined our goal for the project, we agreed that every member of our group should read and understand the project brief until our

first official meeting. In our first meeting we collectively went through the requirements and every single described scenario. After making sure we were all on the same page and understood the content, we defined our goal of the whole project as described in subsection 1.1.
We continued by going through it again and highlighting epics and first user stories. In further cycles we worked on detailing them and lastly started on finding adequate tasks for the now written cards. Those tasks were not yet assigned to individual persons, as we still wanted to have the option to allocate them according to one's time and knowledge on the described topic later on. Soon first challenges arose when it came down to connecting tasks with one adequate user story. It appears that some tasks are used for many user stories, because they describe core functionalities and therefore have to be implemented in order to make the rest working. Then again, sometimes it was just difficult to assign a specific task to a user story at all, because it described a mandatory functionality that just wasn't covered by any adequate user story and creating one seemed not to be possible. We decided to discuss our concerns in the first meeting with the client.
During further discussion we identified the stakeholders of the project as shown in the stakeholder map in figure 2.

As you can see in figure 2 medatixx GmbH & Co. KG is listed as customer and client, since they use the TDA as an inhouse application. The typical users or normal operators of TDA are software testers and developers at medatixx.
In the next step we thought about the boundaries of our system, i.e. the scope of the work. As shown in figure 3, the TDA only has one indirect interface to adjacent systems. That interface is used by medatixx to provide the XML files from which TDA extracts the necessary information. We visualised this connection in the following context diagram.

After we defined the scope of work of the TDA we discussed which architecture and design patterns we could use for our system. Also

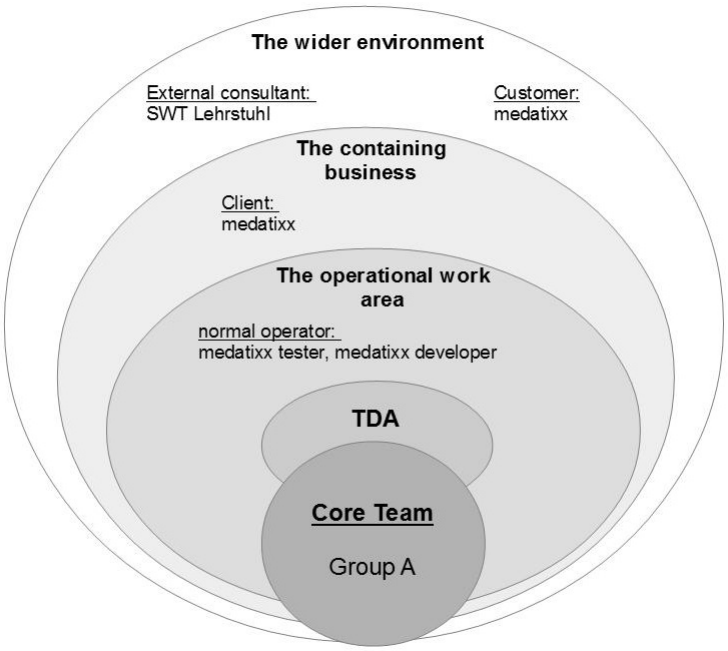

Figure 2: Stakeholder map

first ideas for specific classes and interfaces arose. Gladly, all of us had already visited the DSG-AJP-B course in previous semesters and so we could all contribute equally to the discussion without too much additional explanation of any named techniques. Since we have to deliver a GUI application in Java, we decided on a standard model-view-controller pattern. The corresponding high level architecture diagram is shown below in figure 4.

In the next step we constructed a central glossary to minimize misunderstandings in our communication and make sure to understand the language of the client.

The last task of sprint 0 was to conduct a risk analysis. Most likely we have to deal with sickness of individual persons every now and then. We try to limit the impact of this by good group communication, shared responsibilities and documentation. Also it is likely that

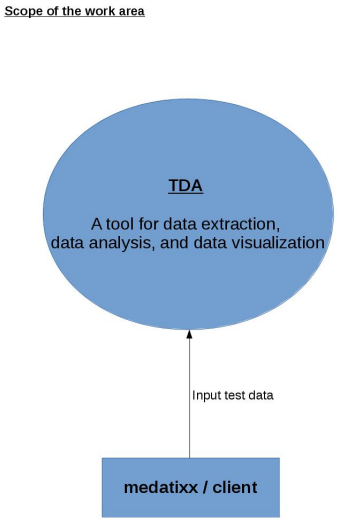

Figure 3: Context diagram

Table 3: Glossary

Term	Meaning
Test run	A collection of unit tests of one specific build
Failure percentage	Failure percentage of class C = (Number of failed unit tests for class C) / (Total number of unit tests for class C)
TDA	Test Data Analyser (the program to be developed)
Problematic class	Class with a high test failure percentage

the client changes the specifications along the way, what we're going to cope by building adaptable software with loose coupling and high cohesion. The risks with the highest impact would be someone leaving the project or the complete loss of all our data. We are going to deal with this with shared responsibilities and backups, respectively. A complete and detailed risk analysis is shown in table 4.

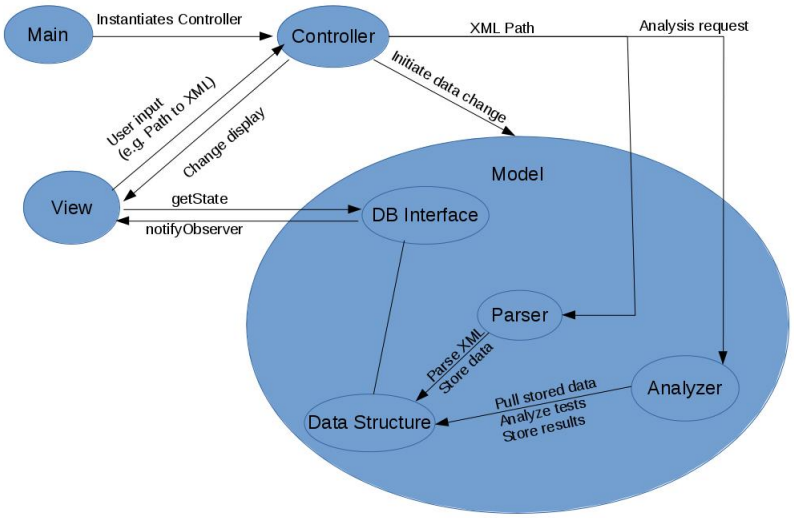

Figure 4: High level architecture

Table 4: Risk analysis

Risk	Coping	Likelyhood	Impact
Someone leaves the project	Shared Responsibilities	<10%	Severe
Complete data loss	Back ups	5%	Severe
Usage of prohibited packages	Regular checks, test on external IDE	10%	High
Sick team member	Shared responsibility, documentation	50%	Medium
Client changes specifications	Adaptable software (loose coupling & high cohesion	50%	Medium
Unprecise specification (missing examples)	Communication with client	20%	Medium
Research stories far more complex than expected	Do research early, conservative planning	5%	Medium
Differing visions (unnecessary development)	Refular internal communication	30%	Medium
Lab PCs insufficient for testing/development	Write performant code	5%	Low

2. Requirements

(Approx. 6–12 pages of text.) Frank

Document and analyse the software's functional requirements, non-functional requirements and development constraints. In particular, state whether a requirement is derived from the project brief, is an assumption made by your group, or has been added by the client. You may apply any documentation and analysis technique taught in module SWT-FSE-B or from the requirements engineering literature, including techniques based on user stories, use cases and prototyping. Properly reference and justify all employed techniques.

This section shall also include a table containing an overview of all user stories. Use Table 5 as a template, and order the stories regarding their ID (story number). Name the source of the story: project brief (PB), the client (C), or other sources. You can use the stories' name+ID in the sequel to refer to a certain story.

Table 5: List of user stories

ID	Name	Size	Source	Sprint
⋮	⋮	⋮	⋮	⋮

2.1. Requirements derived from project brief

2.2. Requirements assumed by us

2.3. Requirements added by client

2.3.1. Functional Requirements

- choose a folder and parse all containing XML files in the folder and its subdirectory.

- Table: classes with failure percentages between 50 - 75% should be highlighted yellow and failure percentages between 75 - 100% should be highlighted red; it should be possible to hide/show all classes with a failure percentage of 0%
- one click in tree sidebars for selecting files (no doubleclick)
- hover over chart entries shall show testrun information
- chart shall have its own page instead of being shown in the main window below the class table
- The functions to filter the Apriori results by confidence and distance should be implemented
- A second additional usage scenario should be proposed, but due to the given point in time the focus should be to finalize the already implemented functionality.

2.3.2. Non-functional Requirements

- Failure percentages in table should be shorted to 2 decimal places
- Testrun totals: a note, stating that not all totals of the testrun are shown, should be visible; a button should enable the user to see all totals.
- program shall adapt its layout automatically to lower resolutions or resizings
- The outcome of the Apriori algorithm should be visualized in a better way, since the two tables are not easy to read.

3. Architecture & Design

3.1. Initial Idea

At the beginning of the project, the model-view-controller architecture has been chosen to be the basis for the TDA. For this, the project is divided into three different layers, each responsible for a specific functionality. In particular, the business logic is separated from the control and presentation layer. This follows the principle of separation of concerns.

The model-view-controller pattern is defined as a compound pattern. This means that it is a combination of several other design patterns that work together. As the name predicts, it consists of a model, view and controller, which all have different responsibilities. The view is responsible for displaying information and for allowing interaction with the user. The model holds the program's data and executes all related computations. It is the representation of the business logic. Finally, the controller works close with the view and functions as a translator for the interaction between the previously mentioned components. View and controller together build the user interface. While the model can function on its own, view and controller depend on the existence of a model. This makes the user interface easily changeable or even exchangeable as a whole, without touching the logic at all. In other words, the proposed architecture is also designed for change. Additionally, it is possible to implement multiple views next to each other, all relying on the same underlying logic. The preferred view can then even be displayed at run time and for example based on the machine or device it is being executed on.

Since the TDA shall display all the information in an adequate graphical user interface, this approach seemed to fulfill the criteria and fit our needs. Although there were no concrete plans for implementing an additional user interface, it is always a good practice to be prepared for similar requests that arise later in the project phase.

Also very early in the project development, we discussed how the given XML-files shall be read and whether they shall be processed and stored internally for later, faster processing. Database approaches were discussed and compared to directly loading the information in an internal infrastructure. A detailed description of this process and the final results can be found in section 3.4. The selected data structure and its location also have a big influence on the selected architecture and the overall class interaction. With the current approach, this clearly was a part of the model.

After other components like the parser and the analyzer were addressed in later project development, the model steadily increased in size and with it in complexity. Other design patterns and architectures were needed, in order to maintain the highly flexible and clearly structured code that we strive to achieve. That's why we changed the high-level architecture, as described in detail in the following paragraph, and used the previously explained model-view-controller pattern only as a subsystem for the new architecture.

3.2. Final approach

The high level architecture of the TDA now follows the repository architecture. This design is defined by one central entity and different subsystems that are connected directly to the named system. The central entity hereby usually functions as a data storage. In the following, the advantages and disadvantages are being weighed against each other and it is concluded why the introduced architecture is a good fit to our needs.

The repository architecture in general is designed for change. Having different subsystems for different functionality allows for exchangeability and therefore flexibility in the later software development and maintenance. Also adding or removing subsystems can easily be done. Furthermore, the central data storage allows for convenient and consistent data management. Data is only stored at one position and is not hold

by any subsystem. This means that changes to the data system are automatically propagated to other components through the shared repository. Subsystems therefore also don't need to worry about how and where the given information is processed and used. They just have to follow the given restrictions on accepted data types.

This also names one of the challenges of the repository architecture. All components need to agree on a certain standard of communication and use the same data types to be compatible. Additionally, one single access point also introduces a single point of failure. It therefore is extremely important that the used system is failure robust and stable over time. The repository should also be able to handle multiple requests at the same time and manage resources efficiently. Since all other systems depend on it, a slowdown would affect the whole project. Finally, the distribution of the repository across many different machines may cause some additional challenges.

It can be concluded that the introduces repository architecture simplifies the access and management of the data in a project significantly. This comes with some restrictions in terms of the communication and data types. Therefore, the proposed architecture is often only a solution for relatively small and structured systems [FSE – V14, p26].

The Test Data Analyzer represents such a small and structured system. Also the project has a limited scope and therefore the final product will remain rather small. Lastly, the repository is well suited for projects with a database. Since the TDA stores data internally, this is fitting too. To concluded, the TDA is benefiting from all of the repository architecture's advantages while limiting the disadvantages to a minimum. Therefore, the proposed architecture is well suited as a high level architecture for the overall project.

While the internal data structure represents the repository, the other components act as its subsystems. These include the user interface, the parser and the analyzer.

3.3. General Principles

3.4. Data Structure

4. Realization

4.1. Sprint Overview

In this subsection a quick overview of all Sprints is given, including the vision underlying each Sprint.
By the end of Sprint 1 TDA should be able to parse the XML files provided by the client into our internal data structure. Furthermore a first paper prototype of TDA should be created to demonstrate our vision of TDA to the client. This way we wanted to make sure, that we understood the project and that he approves of our design decisions.
In Sprint 2 our goal was to display the information we parsed into our data structure in a GUI. The user should be able to select one or more XML files and the TDA presents a table of one chosen test run, containing all tested classes and their failure percentage. The table is sorted in decreasing order of the failure percantage and the classes with the highest failure percentage are highlighted.
During Sprint 3 our goal was to display a chart that visualises the evolution of the tailure percentage of a class over all loaded test runs. Furthermore we wanted to implement the Apriori algorithm to perform a dependency analysis on failed classes. During this Sprint we received the change request from the client which had to be worked into our system.
In Sprint 4 we wanted to rework the Apriori algorithm, since our first implementation was not very efficient. We also wanted to display the results of the Apriori algorithm in our GUI. Furthermore the implementation of our first additional usage scenario should be finished. The additional functionality should enable the user to compare a class in 2 different testruns and see how the outcome of its unit tests have changed.
During Sprint 5 our goal was to finalise our system so that we could present a rounded and finished product to the client. We also wanted to come up with an additional usage scenario for the client, which enables him to further analyse his test runs.

4.2. Sprint No. 1

(Approx. 2–3 pages of text.) Andy

Sprint Planning

State the goal of and the user stories chosen for this sprint (sprint backlog). Detail the tasks that your group derived from each user story, and provide the names of the team members allocated to each task.

Noteworthy Development Aspects

Describe and justify the development approach taken and the artefacts produced in this sprint (e.g., prototypes). State any peculiarities of this sprint, such as peculiarities regarding (i) adopted development practices, (ii) encountered obstacles, (iii) questions that arose and needed clarification possibly from the client, or (iv) important aspects regarding — or changes to — your software architecture, your algorithms or your techniques applied to solve a technical problem.

Sprint Review

Describe the product increment produced in this sprint. Compare the achieved increment with the sprint goal and the user stories that were chosen for this sprint. Give a brief summary on your group's retrospective, including changes to the product backlog and also to the development process and/or techniques that you installed after the sprint in order to overcome any identified obstacle.

4.3. Sprint No. 2

(Approx. 2–3 pages of text.)

Sprint Planning

Noteworthy Development Aspects

Sprint Review

4.4. Sprint No. 3

(Approx. 2–3 pages of text.) Jan

Sprint Planning

The goal for Sprint 3 was to finish and merge the LineChart visual output that missed the Sprint 2 deadline, as well as the implementation the Apriori Algorithm. For the purpose of creating a working Apriori Implementation, a research story with unknown size was assigned.

Noteworthy Development Aspects

Sprint Review

The original intent for Sprint 3 was to implement both the LineChart visual output that missed the Sprint 2 deadline and the Apriori Algorithm. For this purpose, development time was allotted to research , and some trial and error was expected on both user stories. While the GUI implementation of the Chart proved to be easier than expected, the Apriori development did not. Mid Sprint, the Group received the Change Request, and, after analysing the new requirements, it was evident that the planned Apriori implementation would have to be significantly changed after the Change request would be merged in. Thus we decided to prioritize the implementation of the treeview necessary for the adapted Apriori both in the program logic and the GUI.

4.5. Sprint No. 4

(Approx. 2–3 pages of text.)

Sprint Planning

Noteworthy Development Aspects

Sprint Review

4.6. Sprint No. 5

(Approx. 2–3 pages of text.)

Sprint Planning

Noteworthy Development Aspects

Sprint Review

5. Quality Assurance

(Approx. 4–7 pages of text.)

Describe and justify the different quality assurance techniques that your group has applied alongside the project's conduct, including the INVEST criteria for the user stories, SMART criteria for the tasks derived from user stories, unit tests for your code, and others. Illustrate your approach to quality assurance by giving relevant examples for each employed technique. Finally, do not forget to evaluate your software's interfaces (including the GUI).

6. Project Review

(Approx. 3–5 pages of text.) Simon

6.1. Development Process

How well did your group's development process work, and why? Did the process change between sprints? In addition, compare and contrast the SCRUM process as practised by your group to (i) 'the' textbook SCRUM process [?] and (ii) the other software development processes presented in module SWT-FSE-B. Could your group's development process be improved, and by which means?

6.2. Team Work

How well did your team work together? Was the distribution of work and the communication among team members effective? Was the communication with the client effective?

6.3. Lessons Learned

What would you change if you could re-start the project, regarding the employed techniques, the conduct of the project and any other matters that you consider relevant? What should stay the same?

A. Product Backlog

*Insert the final product backlog that includes **all** user stories of your project (cf. front and back sides of your story cards). Order the stories in the backlog regarding the sprint in which they were completed.*

Stories completed in Sprint 1

Include stories that were completed in the first sprint.

Stories completed in Sprint 2

Include stories that were completed in the second sprint.

Stories completed in Sprint 3

Include stories that were completed in the third sprint.

Stories completed in Sprint 4

Include stories that were completed in the fourth sprint.

Stories completed in Sprint 5

Include stories that were completed in the fifth sprint.

Not completed Stories

Include stories that were not completed by the end of the project.

Other Stories

Include here stories that were split or combined and do not appear above.

B. Additional Material

If needed, insert any additional material, e.g., larger diagrams or longer excerpts of source code, in this and possibly further appendices. Properly reference all appendices from the report's main part.

Ehrenwörtliche Erklärung

Alle Unterzeichner erklären hiermit, dass sie die vorliegende Arbeit (bestehend aus dem Projektbericht sowie den separat abgelieferten digitalen Werkbestandteilen) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

- Frank Keßler

Ort/Datum Unterschrift

- Andreas Köllner

Ort/Datum Unterschrift

- Jan Martin

Ort/Datum Unterschrift

- Simon Meyer

Ort/Datum Unterschrift

- Tobias Schwartz

Ort/Datum Unterschrift