MASTER'S THESIS

Software Architecture Analysis Tool

by

Johan Muskens

 Supervisor: Michel Chaudron

Advisors: Rob Westgeest
          Bjorn Bon

Eindhoven, April 2002
Preface

During my study computing science at the Technische Universiteit Eindhoven I worked as an operator at "Bureau Computer Faciliteiten". As an operator I met a lot of scientific employees of the department Computing Science. One of these employees was Michel Chaudron.

Michel asked me whether I was interested in a graduation project concerning component-based software architectures. At the time that was a bit premature, but one year later when I was ready to start my graduation assignment I remembered that conversation.

I made an appointment. We found a third party also interested in software architectures (CMG Eindhoven) and made an assignment description. The assignment was to evaluate component-based software architectures on non-functional attributes.

I would like to thank Michel Chaudron, Rob Westgeest and Bjorn Bon for their guidance and for being my advisors during this project. I would like to thank Gert Florijn and Andre Postma for the useful tips they gave me. Furthermore I would like to thank Jan Friso Groote and Kees Huizing for taking place in my examining board.

Johan Muskens
Department of Mathematics and Computing Science
Technische Universiteit Eindhoven
# Table of Contents

Preface .......................................................................................................................... 3  
Table of Contents ......................................................................................................... 5  
List of Figures ............................................................................................................... 7  
Abstract ........................................................................................................................ 9  
Terms and Abbreviations ............................................................................................ 10  
1 Introduction ............................................................................................................... 11  
2 Software Architecture Analysis .............................................................................. 13  
   2.1 What Is a Software Architecture? ..................................................................... 13  
   2.2 What Do We Analyse of Software Architectures? ....................................... 14  
   2.3 How Do We Analyse Software Architectures? ........................................... 14  
   2.4 Software Architecture Analysis in this project ......................................... 15  
3 "The '4+1' View Model of Software Architecture" ............................................. 16  
   3.1 The Views ....................................................................................................... 17  
      3.1.1 Logical View ............................................................................................ 17  
      3.1.2 Process View .......................................................................................... 19  
      3.1.3 Development View ............................................................................... 19  
      3.1.4 Physical View ....................................................................................... 19  
      3.1.5 Scenarios ............................................................................................... 20  
   3.2 Data Model ....................................................................................................... 21  
4 Metrics ...................................................................................................................... 25  
   4.1 Number of Components per Use-Case ......................................................... 26  
   4.2 Number of Use-Cases per Component ......................................................... 27  
   4.3 Number of Called Services per Component ............................................... 28  
   4.4 Number of Service-Calls of Component ..................................................... 29  
   4.5 Number of Distinct Service-Calls of Component ....................................... 30  
   4.6 Number of Services of Component .............................................................. 31  
   4.7 Coupling ......................................................................................................... 32  
   4.8 Inverse Coupling ............................................................................................ 33  
   4.9 Complexity of Services .................................................................................. 34  
   4.10 Lack of Cohesion between Services ........................................................... 35  
   4.11 Depth of Scenario ......................................................................................... 36  
5 How Do We Interpret the Results? ........................................................................ 37  
6 The Environment of the Analysis Tool ................................................................ 38  
7 The Design of the Analysis Tool .......................................................................... 39  
8 The Implementation of the Analysis Tool ............................................................. 46  
9 The Results .............................................................................................................. 48  
   9.1 MHP Architecture ......................................................................................... 49  
   9.2 Elevator Architecture ...................................................................................... 50  
   9.3 Management Application .............................................................................. 51  
   9.4 Expresso Architecture .................................................................................... 52  
   9.5 Order-System Architecture .......................................................................... 53  

Version: 1.10
Date: 22 March 2002
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Future Work</td>
<td>54</td>
</tr>
<tr>
<td>11 Conclusions and Recommendations</td>
<td>55</td>
</tr>
<tr>
<td>Literature</td>
<td>56</td>
</tr>
<tr>
<td>Books</td>
<td>56</td>
</tr>
<tr>
<td>Papers</td>
<td>56</td>
</tr>
<tr>
<td>Appendices</td>
<td>57</td>
</tr>
<tr>
<td>A Code</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1 - The ‘4+1’ view model................................................................. 16
Figure 2 - Logical view elements ............................................................... 17
Figure 3 - Example of component diagram.................................................. 18
Figure 4 - State diagram elements ............................................................. 18
Figure 5 - State-transition diagram example............................................... 19
Figure 6 - Scenario example........................................................................ 20
Figure 7 - Use case example......................................................................... 20
Figure 8 - Diagram relations......................................................................... 21
Figure 9 - ER diagram.................................................................................. 22
Figure 10 - Number of components for use case.............................................. 26
Figure 11 - Number of use cases for component............................................. 27
Figure 12 - Number of called services of component...................................... 28
Figure 13 - Number of service calls of Component......................................... 29
Figure 14 - Number of distinct service calls of component.............................. 30
Figure 15 - Number of services of component............................................... 31
Figure 16 - Coupling of component............................................................... 32
Figure 17 - Inverse coupling of component.................................................... 33
Figure 18 - Complexity of services ............................................................... 34
Figure 19 - Lack of cohesion between services.............................................. 35
Figure 20 - Depth of scenario....................................................................... 36
Figure 21 - Environment of analysis tool....................................................... 38
Figure 22 - Use case diagram....................................................................... 39
Figure 23 - Component diagram (SAAT)......................................................... 40
Figure 24 - Scenario (Analyse Architecture).................................................. 41
Figure 25 - State diagram (SAAT)................................................................. 42
Figure 26 - State diagram (statistic filter)....................................................... 42
Figure 27 - State diagram (statistic calculator)............................................... 42
Figure 28 - State diagram (Analyser)............................................................. 43
Figure 29 - State diagram (database checker)................................................ 43
Figure 30 - State diagram (database filler)..................................................... 44
Figure 31 - State diagram (database creator).................................................. 44
Figure 32 - State diagram (parser)............................................................... 45
Abstract

The Software Engineering discipline lacks the ability to evaluate software architectures. Software architectures can be evaluated on their functional behaviour as well as on their non-functional behaviour.

With non-functional behaviour of a software architecture we mean quality attributes of a software architecture. For example:

- Performance
- Availability
- Scalability
- Flexibility
- Reusability
- Extendibility

This document considers the creation of a Software Architecture Analysis Tool to evaluate a software architecture on its non-functional characteristics.
# Terms and Abbreviations

<table>
<thead>
<tr>
<th>ATAM</th>
<th>Architectural Tradeoff Analysis Method. Different techniques for analysing architectures with respect to various quality attributes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ER-Diagram</td>
<td>Entity Relation diagram. Diagram used for modelling data.</td>
</tr>
<tr>
<td>Quality attribute</td>
<td>Non-functional characteristic of a software architecture (for example: reusability, scalability, performance, etc).</td>
</tr>
<tr>
<td>SQL</td>
<td>Standard Query Language. Language for adding, modifying, removing and retrieval of data from a database management system.</td>
</tr>
</tbody>
</table>
1 Introduction

The first step in making good software is making a good design. It's the first stage in software development. The design defines the architecture of the software to be built. The quality of the software highly depends on the architecture defined in the early stages of the development process.

The architecture can influence the functional requirements as well as the non-functional requirements. The impact of architectural design decisions in a software development process is very high. The correction of bad / wrong design decisions takes a lot of effort. Therefore it's useful to be able to analyse software architectures in an early stage of the software development process.

At the present time there are a few methods to evaluate software architectures. SAAM and ATAM are by far the most well known. All these methods are evaluation techniques of quality attributes of a software architecture by (a group of) architects.

This project considers the creation of a software architecture analysis tool. This tool should help an architect in evaluating and improving his design. The algorithms used should give more insight in the relation between quality attributes and certain design properties. These insights should help improving a design.

The goal of the Software Architecture Analysis Tool (SAAT) is not to state whether a design is good or bad. The tool gives an indication which elements of a design should be reconsidered. In the end it's always the architect that will decide whether adjustments have to be made or some elements have to be redesigned.

The Software Architecture Analysis Tool must make quantitative statements on quality attributes of a software architecture. The tool must give an indication on problem elements in software architectures. Suggesting improvement on the software architecture is beyond the scope of this project.

The tool must be able to analyse architectures described in the UML modelling tool Rational Rose. The tool must be easy to use and easily extendible with more analysis methods.
This document considers the creation of the Software Architecture Analysis Tool and structured as follows:

Chapter 2 discusses software architecture analysis. It describes what software architectures are (section 2.1), what can be analysed of software architectures (section 2.2), how software architectures can be analysed (section 2.3) and the used approach in this project (section 2.4). I use metrics based on use-cases and scenarios in my analyses.

Chapter 3 introduces the notation used to describe software architectures during this project (The '4+1' View Model of Software Architecture) and the data model used to represent this software architecture description.

Chapter 4 discusses software architecture metrics and gives examples of metrics based on "The '4+1' View Model of Software Architecture". These are the metrics implemented by the Software Architecture Analysis Tool.

Chapter 5 concerns the interpretation of metrics. The results of metrics are tables with scores (numbers) of elements in the architecture (for example components, scenarios and use-cases). This chapter describes how to identify problem elements based on these tables.

Chapter 6 describes the environment of the Software Architecture Analysis Tool.

Chapter 7 contains the design of the Software Architecture Analysis Tool.

Chapter 8 discusses the implementation of the Software Architecture Analysis Tool.
Appendix A: Code contains the full code of the tool.

Chapter 9 discusses the results of analyses of a small number of software architectures with the Software Architecture Analysis Tool.

Chapter 10 contains a list of ideas for improvement or extension of the Software Architecture Analysis Tool.

Chapter 11 contains my conclusions and recommendations.
2  Software Architecture Analysis

This section discusses the analysis of software architectures. But before we can do this we must make clear what we mean with "software architecture".

2.1  What Is a Software Architecture?

There are several definitions of software architectures, almost as many as there are software architects. Here are some examples.


The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction.

(Booch, Rumbaugh, and Jacobson, 1999)

An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition.

During this project we use the first definition. For the analysis of an architecture the definition is not the most important thing. The notation used to describe the architecture is more important. We use “The ‘4+1’ View Model of Software Architecture” by Phillippe Kruchten, which is described and explained in chapter 8 of this document, to describe a software architecture.
2.2 What Do We Analyse of Software Architectures?

Software architectures can be analysed on functional requirements as well as non-functional requirements. This project considers the analyse of software architectures on non-functional quality attributes like extendibility, maintainability, scalability, reusability, etc.

Software architectures can be analysed on several aspects. These aspects can be divided into the following categories:

- **Structural**: The structural aspects of an architecture are described in the logical view. Logical decomposition of the system in components and distribution of services over the components are some key examples of structural aspects.

- **Behavioural**: The behavioural aspects of an architecture are described in the process view, which considers the behaviour of the components.

- **Semantical**: Semantical aspects are not described in any kind of diagram, but concern the meaning / interpretation of the software architecture description.

2.3 How Do We Analyse Software Architectures?

Software architectures can be analysed in several ways.

- **Metrics**: Metrics can give useful information on an architecture, however the interpretation of metrics can be quite complicated. It can give an indication of the dependencies between architecture elements, size of an architecture, complexity of an architecture, etc.

- **Conformance to patterns / styles**: Pattern and styles are often used in software architecture designs. The represent a simple concept and have proven their value over time. Therefore it can be useful to check whether an architecture conforms to a style or pattern.

- **Bottlenecks and anti-patterns**: Bottlenecks and anti-patterns are typically things an architect wants to avoid in his design. A tool that points the bottlenecks / anti patterns in a design can be useful.

- **Use cases & scenarios**: An architecture can be studied by scenarios / use cases. Use cases / scenarios are the requirements of an software architecture. They can give information on which architecture elements are logically related, which helps in the analysis of an architecture.
2.4 Software Architecture Analysis in this project

The categorisations mentioned in section 2.2 (What) and 2.3 (How) are illustrated in the next table. Note that some categorisations are arbitrary and that the table is not complete.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Conformance of pattern / style</th>
<th>Bottlenecks and anti patterns</th>
<th>Use cases &amp; scenarios</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structural</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Degree of coupling</td>
<td>Layers</td>
<td>#services to big</td>
<td>Cohesion</td>
</tr>
<tr>
<td>Degree of cohesion</td>
<td>Pipe filter</td>
<td>(god class)</td>
<td>Depth of scenario</td>
</tr>
<tr>
<td>Directed connectivity</td>
<td>Bus</td>
<td>#services to small</td>
<td>#components involved</td>
</tr>
<tr>
<td>Fan in / fan out Cycles</td>
<td>Blackboard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complexity of serv.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Behavioural</strong></td>
<td>#outgoing calls /</td>
<td>Deadlock</td>
<td>Reachability</td>
</tr>
<tr>
<td></td>
<td>#incoming calls</td>
<td>State-trap</td>
<td>Coverage</td>
</tr>
<tr>
<td></td>
<td>#active components</td>
<td></td>
<td>Disjunct. gr. o. services</td>
</tr>
<tr>
<td><strong>Semantical</strong></td>
<td>Get/Set</td>
<td></td>
<td>Unrelated services within 1 comp.</td>
</tr>
</tbody>
</table>

During this project the analysis methods based on use-cases and scenarios have the main focus, because they look the most promising and are the most innovative.
3 "The ‘4+1’ View Model of Software Architecture"

Software architecture deals with abstraction, with decomposition and composition, with style and esthetics. To describe a software architecture I use "The ‘4+1’ View Model of Software Architecture" by Philippe Kruchten (Rational Software Corp.) [H]. This model is composed of multiple views or perspectives:

- logical view
- process view
- physical view
- development view

The description of an architecture can be organised around these four views, and then illustrated by use cases, or scenarios which become a fifth view.

![Diagram of the '4+1' view model]

**Figure 1 – The ‘4+1’ view model**

We will now look at the views in turn. Only the "Logical view" and the "Scenarios" are analysed by the Software Architecture Analysis Tool.
3.1 The Views

3.1.1 Logical View

The logical architecture primarily supports the functional requirements – what the system should provide in terms of services to its users. The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain, in the form of components. Besides the functional decomposition in components the logical view shows the logical dependencies of the components.

For the logical view we use a component diagram. The component diagram consists of the following elements:

- components
- relations
- interface descriptions

In this view we consider components to be a black box with an interface. This interface is a list of services which the component provides to the outside world. This list contains for each service the name and the parameter list of the service. A component can use one or more services of another component, this is modelled by the uses relation.

![Component Diagram](image.png)

_Figure 2 – Logical view elements_

The notation used in the logical view is illustrated above. How these elements are used to create a component diagram is illustrated by the following example.
In addition to the decomposition in components we describe the behaviour of the components in a state-transition diagram, which has the following elements:

- State
- State-transition
- Begin state indication

The states show all the different states of a component, the state-transitions show the allowed state changes of a component and the begin state shows the first state of a component (when it is created).
3.1.2 Process View

The process view takes some non-functional requirements into account, such as performance and availability. It addresses issues of concurrency and distribution, of system integrity, of fault-tolerance, and how the main abstractions from the logical view fit in the process architecture.

3.1.3 Development View

The development view focuses on the actual software module organisation in the software development environment. The software is packaged in small chunks – program libraries, or subsystems – that can be developed by one or a small number of developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above.

3.1.4 Physical View

The physical view primarily takes into account the non-functional requirements of the system such as availability, reliability, performance and scalability. The software executes on a network of computers, or processing nodes. The various elements identified – networks, processes, tasks and objects – need to be mapped onto the various nodes. Several different physical configurations can be used: some for development and testing, others for the deployment of the system for various sites or for different customers. The mapping of the software to the nodes therefore needs to be highly flexible and have minimal impact on the source code itself.
3.1.5 Scenarios

The elements in the four views are shown to work together seamlessly by the use of a small set of important scenarios – instances of more general use cases. The scenarios are in some sense an abstraction of the most important requirements.

A scenario is described by a message sequence diagram. These diagrams contain the following elements:

- Components
- Service calls

![Figure 6 - Scenario example](image)

Normally there are several scenarios. Each scenario is an instance of a more generic use case. The use cases are shown in a use case diagram.

![Figure 7 - Use case example](image)

The relation between a scenario and a use case is not explicitly shown in a diagram.
3.2 Data Model

During this project we restrict the analysis of a software architecture to the analysis of the following diagrams (logical view and scenarios).

- Use-case diagram
- Sequence diagrams (scenarios)
- Component diagrams
- State–transition diagrams

These diagrams are related. A use-case diagram is linked to several scenarios. A sequence diagram contains components from the component diagram. The state–transition diagram shows the states and state–transitions of a component. This is illustrated by the following figure.

![Diagram showing relationships between use-case scenario, sequence diagrams, and state–transition diagrams](image)

**Figure 8 – Diagram relations**

In order to be able to analyse these diagrams we distilled an abstract data model from the diagrams. It is obvious that the diagrams contain more information, but the distilled information is sufficient for the calculation of the metrics considered during this project. It is likely that for the calculation of new metrics the following data selection needs to be extended.
- Set of all use-cases
- Set of all scenarios
- Set of all components
- Set of all services
- Set of all states
- Set of all service calls

The elements of these sets are related. For example a use-case has several scenarios, a scenario contains several service calls and a service call has a caller, callee, predecessor and a service that is called. This is modelled in the following Entity-Relation diagram.

![Entity-Relation diagram](image)

**Figure 9 – ER diagram**

This ER diagram results in the following table definitions:
Table U contains all use-cases. For each use-case the NAME and an ID are stored. The NAME is the name of the use-case has in the use-case diagram. The ID is a unique string to identify the use-case.

<table>
<thead>
<tr>
<th>U</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>CHAR</td>
</tr>
<tr>
<td>NAME</td>
<td>CHAR</td>
</tr>
</tbody>
</table>

Table S contains all scenarios. For each scenario the NAME and an ID are stored. The NAME is the name of the sequence diagram describing the scenario. The ID is a unique string to identify the scenario.

<table>
<thead>
<tr>
<th>S</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>CHAR</td>
</tr>
<tr>
<td>NAME</td>
<td>CHAR</td>
</tr>
</tbody>
</table>

Table C contains all components. For each component the NAME and an ID is stored. The NAME is the name the component has in the component diagram. The ID is a unique string to identify the component.

<table>
<thead>
<tr>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>CHAR</td>
</tr>
<tr>
<td>NAME</td>
<td>CHAR</td>
</tr>
</tbody>
</table>

Table M contains all services. For each service the NAME and an ID is stored. The NAME is the name of the service (as described in the component diagram). The ID is a unique string to identify the service.

<table>
<thead>
<tr>
<th>M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>CHAR</td>
</tr>
<tr>
<td>NAME</td>
<td>CHAR</td>
</tr>
</tbody>
</table>

Table T contains all states. For each state the NAME and an ID is stored. The NAME is the name of the state (as described in one of the state-transition diagrams). The ID is a unique string to identify the state.

<table>
<thead>
<tr>
<th>T</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>CHAR</td>
</tr>
<tr>
<td>NAME</td>
<td>CHAR</td>
</tr>
</tbody>
</table>

Table US is used to store the ("has") relation between a use-case and a scenario. The table has two fields. ID_U identifies the use-case and ID_S identifies the scenario.
Table CM is used to store the ("provides") relation between a component and a service. The table has two fields. ID_C identifies the component and ID_M identifies the service.

<table>
<thead>
<tr>
<th>US</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_U</td>
</tr>
<tr>
<td>ID_S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_C</td>
</tr>
<tr>
<td>ID_M</td>
</tr>
</tbody>
</table>

Table SM contains all service calls. For each service call an id, scenario, caller (component), callee (component) and a service is stored. In some cases a predecessor is stored. The table has six fields. ID is a unique string identifying the service call, ID_S identifies the context (scenario) in which the call is made, ID_C1 identifies the caller of the service, ID_C2 identifies the callee, ID_M2 identifies the called service and ID_PRED identifies the service call preceding this call in the specific scenario (if available).

<table>
<thead>
<tr>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>ID_S</td>
</tr>
<tr>
<td>ID_C1</td>
</tr>
<tr>
<td>ID_C2</td>
</tr>
<tr>
<td>ID_M2</td>
</tr>
<tr>
<td>ID_PRED</td>
</tr>
</tbody>
</table>

Table CT is used to store the ("has") relation between a component and a state. The table has two fields. ID_C identifies the component and ID_T identifies the state.

<table>
<thead>
<tr>
<th>CT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_C</td>
</tr>
<tr>
<td>ID_T</td>
</tr>
</tbody>
</table>

Table TT is used to store the ("can change to") relation between two states. The table has two fields. ID_T1 identifies the "from" state and ID_T2 identifies the "to" state.

<table>
<thead>
<tr>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID_T1</td>
</tr>
<tr>
<td>ID_T2</td>
</tr>
</tbody>
</table>
4 Metrics

In this project the analysis of an architecture consists of the calculation of several metrics. For software metrics there are several guidelines. Metrics must have the following properties:

- Simple and computable – the metrics should be easy to learn and use
- Empirically convincing – they should satisfy the expectations of the engineer
- Consistent and objective – they should produce unambiguous results
- Consistent in dimensionality – they should be mathematically reasonable
- Programming language independent
- Facilitating feedback – they should provide useful information for software improvement

Metrics usually are sizes of selections or a simple function based on several selection sizes, which means that the metrics are queries and can be defined in the database query language SQL.

Defining the metrics in SQL has several benefits:

- **Extendibility**: Adding and removing analysis methods / metrics to the analysis tool comes down to executing a new SQL statement or removing an SQL statement. It's possible to read the analysis methods / metrics from an external text file, which makes it possible to configure the analysis tool such that it performs the correct analysis.
- **Easy implementation**: Implementing the calculation of the metrics comes down to the execution of an SQL statement. Execution of SQL statements is possible in many database management systems. One of these database management systems can be used for the implementation of the analysis tool.

Several metrics have been implemented in the Software Architecture Analysis Tool. This is an arbitrary selection. Due to the time limit on this project it is not possible to implement all the metrics. In the next sections the metrics implemented by the Software Architecture Analysis Tool are described.
4.1 Number of Components per Use-Case

Use-cases describe the functional requirements of a system. Therefore a use-case can be considered as a requirement or a selection of related requirements. The requirements are implemented by components. When the number of components used for implementing a use-case is high, this means that changing a use-case (requirement) can have impact on a large number of components. More specific it means that related functionality is spread over the design.

This metric counts the components that call a service or from which a service is called in context of a scenario that is part of a specific use-case. The main thought behind this metric is that related functionality spread over the design is bad for maintainability and reusability, because:

- Reuse of functionality then means the reuse of a large number of components.
- Understanding how functionality is implemented, then requires knowledge on a large number of components.

![Diagram](image)

**Figure 10 - Number of components for use case X = 7**

Definition, using the abstract data model described in section 3.2:

\[
\forall c, id_u \in c : (\exists id_{id_u}, id_{id}, id_{id_u}, id_{id_u}, id_{id_u} : ((id_{id} = c.id \lor id_{id_u} = c.id) \land (id_{id_u}, id_{id_u} \in Us)))
\]

Query, using the abstract data model described in section 3.2:

```sql
SELECT U.name, COUNT(DISTINCT C.id) FROM U, US, SM, C
WHERE (C.id = SM.id_c1 OR C.id = SM.id_c2) AND (SM.id_s = US.id_s)
AND (U.id = US.id_u) GROUP BY U.id;
```
4.2 Number of Use-Cases per Component

When a component is used for the implementation of several use-cases it is likely that the component implements requirements that are logically unrelated. It usually means that the cohesion between the different services of the component is low.

This metric counts the number of use-cases that contain a scenario in which a service of a specific component is called or in which that component calls a service. The main thought behind this metric is that cohesion within a component should be high (Number of use-cases per Component should be low). High cohesion within a component is good for maintainability, because:

- This means the component has implements logical dependent functionality.

![Diagram of use cases](image)

Figure 11– Number of use cases for component

Number of use cases for component X = 2

Definition, using the abstract data model described in section 3.2:

\[
\text{(\# u.id \in U : (\exists v.id \in v.id, u.id \in u.id \text{ and } id \in id \in u.id \implies \text{ sm\{\{(id\_c1 = c.id \lor id\_c2 = c.id) \land (id\_u, id\_s \in Us)\}}

\text{ for all } c.id}
\]

Query, using the abstract data model described in section 3.2:

```sql
SELECT C.NAME, COUNT(DISTINCT US.ID_U) FROM C,US,SM
WHERE (C.ID = SM.ID_C1 OR C.ID = SM.ID_C2)
    AND (SM.ID_S = US.ID_S) GROUP BY C.ID;
```
4.3 Number of Called Services per Component

Tasks should be distributed over the design as equally as possible. When the number of called services of a component is high this can give an indication that the component is a possible bottleneck when considering scalability. It also indicates that dependency of other components on the specific component is high.

This metric counts the number of called services of a specific component for all scenarios.

![Diagram showing the number of called services for components A, B, C, and X.]

**Figure 12 – Number of called services of component**

Definition, using the abstract data model described in section 3.2:

\[
\{ \text{id.c} \} \text{.id_{c1}.id_{c2}.id_{c3}.id_{c4}.id_{c5} \text{ for } c \text{ in sm: id.c2 = c.id} \text{ for all c.id} }
\]

Query, using the abstract data model described in section 3.2:

SELECT C.NAME, COUNT(*) FROM C,SM WHERE (C.ID = SM.ID_C2) GROUP BY C.ID;
4.4 Number of Service-Calls of Component

A component usually uses services of other components. This means that the component is dependent on the other component. The dependence on other components increases with the number of service calls of a component.

This metric counts the number of service calls of a specific component for all scenarios. Main thought behind this metric is that dependencies are bad for reusability and maintainability, because:

- Reuse of a specific component then require reuse of a large number of components.
- Maintenance of a specific component then requires knowledge of a large number of other components.

![Diagram of service calls between components X, A, B, and C]

Number of service calls of component X = 5

Figure 13 – Number of service calls of Component

Definition, using the abstract data model described in section 3.2:

\[ \forall c_\text{id}, c_\text{id}_\text{c1}, c_\text{id}_\text{c2}, c_\text{id}_\text{pred} \in \text{sm}: c_\text{id}_\text{c1} = c_\text{id} \text{ for all } c_\text{id} \]

Query, using the abstract data model described in section 3.2:

```
SELECT C.NAME, COUNT(*) FROM C.SM WHERE (C.ID = SM.ID_C1) GROUP BY C.ID;
```
4.5 Number of Distinct Service-Calls of Component

A component usually uses services of other components. This means that the component is dependent on the other component. The dependence on other components increases with the number of service calls of a component.

This metric counts the number of different services called by a specific component for all scenarios. Main thought behind this metric is that dependencies are bad for reusability and maintainability, because:

- Reuse of a specific component then require reuse of a large number of components.
- Maintenance of a specific component then requires knowledge of a large number of other components.

Figure 14 – Number of distinct service calls of component

Definition, using the abstract data model described in section 3.2:

\[
\# \text{id} \in \{3 \text{id.id} \land \text{id.c1.id} \land \text{id.m2.id} \land \text{id.m2.id} \land \text{id.c1.id} = \text{id.c1.id} \land \text{id.m2.id} = \text{id.m2.id}\} \text{ for all c.id}
\]

Query, using the abstract data model described in section 3.2:

```
SELECT C.NAME, COUNT(DISTINCT SM.ID_M2) FROM C,SM
WHERE (C.ID = SM.ID_C1) GROUP BY C.ID;
```
4.6 Number of Services of Component

A component provides services. It is wise to distribute functionality evenly over the design. Large differences in the number of services per component can give an indication that this is not the case.

This metric counts the number of provided services of a specific component. The main thought behind this metric is that a well balanced distribution of functionality over the design is good for extendibility and maintainability, because:

- Excessively large component are difficult to understand.

![Diagram of Component X with services](image)

Figure 15 - Number of services of component

Definition, using the abstract data model described in section 3.2:

\[
\# \langle id\_m, id\_c \rangle \in \mathcal{N} : (\exists \langle id\_m, id\_c \rangle \in \mathcal{N} : id = id\_m \land id\_c = c.id) \text{ for all } c.id
\]

Query, using the abstract data model described in section 3.2:

```sql
SELECT C.NAME, COUNT(DISTINCT CM.ID_M) FROM C, CM
WHERE (C.ID = CM.ID_C) GROUP BY C.NAME;
```
4.7 Coupling

A component uses services of other components, this means the component is dependent on the other component.

This metric counts the number of components of which a service is called. The main thought behind this metric is that dependence on many different components is bad for reusability, extendibility and maintainability, because:

- Reuse of a specific component then require reuse of a large number of components.
- Maintenance of a specific component then requires knowledge of a large number of other components.

![Diagram of component coupling]

**Figure 16 – Coupling of component**

Definition, using the abstract data model described in section 3.2:

\[ \text{coupling} = \sum_{c1} \exists c_2 \text{ such that } c_1 \text{ and } c_2 \text{ are connected} \]

Query, using the abstract data model described in section 3.2:

```sql
SELECT C.NAME, COUNT(DISTINCT SM.ID_C2) FROM C, SM
WHERE (C.ID = SM.ID_C1) GROUP BY C.ID;
```
4.8 Inverse Coupling

Services of a component are used by other components, this means that the other components depend on the component.

This metric counts the number of components that call any service of a specific component. The main thought behind this metric is that a large number of components depending on a specific component is bad for extendibility and maintainability, because:

- Changes in a specific component can then influence the behaviour of many other components.

![Diagram showing inverse coupling](image)

Inverse coupling (of component X) = 3

**Figure 17 – Inverse coupling of component**

Definition, using the abstract data model described in section 3.2:

\[ \# \text{id.name} \in C \ (3 \text{id.id}_x, \text{id}_c1, \text{id}_c2, \text{id}_c3, \text{id}_p1, \text{id}_p2) \in SM : \text{id}_c2 = \text{c.id} \land \text{id}_c1 = \text{id}) \text{for all c.id} \]

Query, using the abstract data model described in section 3.2:

```sql
SELECT C.NAME, COUNT(DISTINCT SM.ID_C1) FROM C,SM
WHERE (C.ID = SM.ID_C2) GROUP BY C.ID;
```
4.9 Complexity of Services

This metric is an attempt to give an indication of the average complexity of the services of a component. A component has a number of states and transitions between these states. Service executions are responsible for state changes (transitions). Therefore we presume that services of a component are more complex when they are responsible for a larger number of state changes.

This metric computes the average number of state transitions per service for a component. Main thought behind this metric is that complexity of components / services is bad for maintainability and extendibility.

![Diagram of Component X with state transitions]

Complexity of services = 7 / 2

Figure 18 – Complexity of services

Definition, using the abstract data model described in section 3.2:

(# \( \langle \text{id}_c, \text{id}_d \rangle \) \( \text{et}_c = \text{true} \)) / (# \( \langle \text{id}_c, \text{id}_d \rangle \) \( \text{et}_c : \text{id}_c = \text{c.id} \)) for all c.id

Query, using the abstract data model described in section 3.2:

```
SELECT C.NAME, COUNT(DISTINCT TT.ID_T1, TT.ID_T2) / COUNT(DISTINCT CM.ID_M)
FROM C,CM,CT,TT
WHERE (C.ID = CM.ID_C AND C.ID = CT.ID_C AND CT.ID_T = TT.ID_T1)
GROUP BY C.NAME;
```
4.10 Lack of Cohesion between Services

This metric is related to the number of use-cases for which a component is used. It also gives an indication of the cohesion between the services provided by a component, it indicates whether the different services of a component are logically related. This metric counts the maximum number of subsets of services of a component such that the sets of use-cases using services of the subsets are disjoint. This is illustrated by the following diagram.

![Diagram illustrating lack of cohesion between services]

**Figure 19 – Lack of cohesion between services**

Definition, using the abstract data model described in section 3.2:

\[ M_c(c) = \{ \langle id, name \rangle \in M | \langle id, c, id, m \rangle \in C_m \} \]

\[ S_n(m) = \{ \langle id, name \rangle \in S | (\exists \langle id, s, id, c, id, m, id, name \rangle \in S_m : (id, m_2 = m)) \} \]

\[ U_m(m) = \{ \langle id, name \rangle \in U | \langle id_u, id_s \rangle \in Us \land \langle id_s, name \rangle \in S_n(m)) \]

\[ D(M_1, M_2) = (\forall m_1, m_2: m_1 \in M_1 \land m_2 \in M_2: (U_m(m_1) \cup U_m(m_2) = \emptyset)) \]

\[ \frac{\uparrow \forall e (p \in \mathcal{P}(c) \setminus (11)) \lor \forall M_1, M_2 e p \wedge M_1 \neq M_2 : \Delta(M_1, M_2) : |d|) \text{ for all } c.id \]

Query, using the abstract data model described in section 3.2:

```sql
SELECT C.NAME, COUNT(DISTINCT CM.ID_M) - ((COUNT(US.ID_U) - COUNT(DISTINCT US.ID_U)) - (COUNT(CM.ID_M) - COUNT(DISTINCT CM.ID_M, US.ID_U)))
FROM C, CM, US, SM
WHERE (C.ID = CM.ID_C) AND (CM.ID_M = SM.ID_M2) AND (US.ID_S = SM.ID_S)
GROUP BY C.NAME;
```
4.11 Depth of Scenario

Keep software architectures as simple as possible. Simplicity is good for maintainability and reusability. This metric gives an indication of the complexity of a scenario. It measures how deep the service calls are nested for a scenario. If the depth of a scenario is too high this is bad for the understandability and therefore also bad for maintainability and adaptability. Note that this metric does not necessarily indicate the complexity of an architecture, it indicates the complexity of some of the diagrams used to describe the architecture.

This metric measures how deep the service calls are nested for a scenario.

![Diagram showing depth of scenario]

Figure 20– Depth of scenario

Definition, using the abstract data model described in section 3.2:

\[
\forall \langle id, name \rangle \in c \ (\exists \langle id, id_c, id_c1, id_c2, id_pred, id \rangle \in sm: id_c1 = id \land id_s = s.id) \text{ for all } s.id
\]

Query, using the abstract data model described in section 3.2:

```sql
SELECT S.NAME, COUNT(DISTINCT SM.ID_C1) FROM S, SM
WHERE (S.ID = SM.ID_S) GROUP BY S.NAME;
```
5 How Do We Interpret the Results?

The metrics described in the previous sections give values for certain elements (use-cases, scenarios or components). The result of the query is a table containing two columns. Each record has an element description (first field) and a value (second field). What can we do with this table?

During this project we use the following approach. We do not use benchmark values telling whether a score of an element is good or bad, but we compare the score of an element with the score of the other elements within the design. We look for the elements that have outlying values, because we suspect these elements to be the problem elements.

Now the only problem is how to figure out what the outlying values are. We consider values that differ more than 2 times the standard deviation from the mean value to be outlying values. The standard deviation is calculated as follows:

First calculate the variance $\sigma^2$.

$$\sigma^2 = \frac{\sum(x - \mu)^2}{N}$$

Were $\mu$ is the mean value of the distribution (all scores of the elements) and $N$ is the number of scores. The standard deviation is the square root of the variance. It is the most commonly used measure of spread.

Consider the following example of output of a metric.

<table>
<thead>
<tr>
<th>Component</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component A</td>
<td>2</td>
</tr>
<tr>
<td>Component B</td>
<td>3</td>
</tr>
<tr>
<td>Component C</td>
<td>3</td>
</tr>
<tr>
<td>Component D</td>
<td>2</td>
</tr>
<tr>
<td>Component E</td>
<td>1</td>
</tr>
<tr>
<td>Component F</td>
<td>15</td>
</tr>
<tr>
<td>Component G</td>
<td>2</td>
</tr>
<tr>
<td>Component H</td>
<td>3</td>
</tr>
</tbody>
</table>

Average: 3.875  
Standard deviation: 4.26

Conclusion based on the average and standard deviation:  
**Component F** has an outlying value and is a possible problem element.

Note that not all elements that have an outlying score are design errors. It is up to the architect to judge whether an element is a real problem and redesign is necessary. However the better the algorithm to find the outlying values, the bigger the added value of the Software Architecture Analysis Tool.
6 The Environment of the Analysis Tool

SAAT analyses architectures created with Rational Rose (UML modelling tool). During this project Rational Rose is used for the following tasks:

- Creation of the software architectures
- Export of software architectures to an interchange file

Input for SAAT is the interchange file created by Rational Rose. SAAT has one specific task: the creation of an analysis report. But before this report can be created the architecture has to be stored in a database and analysed.

The output of SAAT is a report in HTML format. This report can be read with an ordinary HTML browser.

Figure 21- Environment of analysis tool
7 The Design of the Analysis Tool

The Software Architecture Analysis Tool has one specific task: analysis of software architectures. The user starts the application and the analysis tool produces a report containing the results of the analysis. Therefore the tool has only one use-case. This is illustrated by the following use-case diagram.

![Use case diagram]

**Figure 22 – Use case diagram**

The tool consists of several components working together. Each component has its own specific responsibilities. The components are:

- **Parser**: This component extracts the relevant architecture information from the input file (use-cases, scenarios, components, states, services and their relations). The input file is an .xmi file generated by Rational Rose. Output of the parser is plain ASCII. The output of the parser is used by the database filler.
- **Database creator**: This component creates a new database with the database management tool (MySQL) and creates empty tables in this database which will be used to store the architecture information.
- **Database filler**: This component fills the database, created by the database creator, with the software architecture information extracted from the .xmi file by the parser.
- **Database checker**: This component checks the database for incomplete information. It checks whether all components have a state diagram, whether all use cases have at least one scenario, etc.
- **Analysers**: This component executes the actual architecture analysis. It analyses the architecture information in the database created by the database creator and the database filler.
- **Statistic calculator**: This component calculates some statistics on the results of the software architecture analysis. It calculates the mean value, number of elements for which a value is calculated and the standard deviation of the values.
Software Architecture Analysis Tool

- Statistic filter: This component filters the result based on the statistics calculated by the statistic calculator such that only the elements with the outlying values remain.
- Saat: This is a control component that is used to configure the Software Architecture Analysis Tool.

This is illustrated by the following component diagram.

![Component Diagram](image)

**Figure 23– Component diagram (SAAT)**
How the components work together is illustrated by the following scenario.

Figure 24 – Scenario (Analyse Architecture)
The individual components can be in states. This is modelled by the following state-transition diagrams.

**Figure 25 – State diagram (SAAT)**

**Figure 26 – State diagram (statistic filter)**

**Figure 27 – State diagram (statistic calculator)**
Figure 28 – State diagram (Analyser)

Figure 29 – State diagram (database checker)
Software Architecture Analysis Tool

Figure 30 – State diagram (database filler)

Figure 31 – State diagram (database creator)
Figure 32 – State diagram (parser)
8 The Implementation of the Analysis Tool

For the implementation of the Software Architecture Analysis Tool several tools and programming languages were used.

The parser extracts architecture information from a .xmi file. XML stands for XML Metadata Interchange. XML provides a standard way to exchange information about metadata between modeling tools based on the UML (unified modeling language) object–based modelling language (for more information see the website: http://www.xml.com/pub/rg/XML.XML_Metadata_Exchange).

The extraction of architecture information concerns a lot of string manipulation and pattern matching therefore (Active) Perl is used for the parser (for more information see the website: http://www.activestate.com/Products/ActivePerl/). All the methods of the component parser are implemented by a Perl script. These scripts are:

- ExtractUseCases.pl
- ExtractScenarios.pl
- ExtractComponents.pl
- ExtractStates.pl
- ExtractServices.pl
- ExtractUs.pl
- ExtractSm.pl
- ExtractCm.pl
- ExtractCt.pl
- ExtractTt.pl

All these scripts write the results to an output file. The output of the parser is comma separated ASCII.

The database creator is a Visual C++ application that uses a MySQL library to create a new database.

The database is filled by the database filler. This consists of several Perl scripts that process the output of the parser and store the elements in the MySQL database. This is done by calling a Visual C++ application that executes SQL statements. The Perl scripts are:

- FillU.pl
- FillS.pl
- FillC.pl
- FillT.pl
- FillM.pl
- FillUS.pl
- FillSM.pl
- FillCT.pl
• FillCM.pl
• FillIT.pl

The database checker is a Perl script that reads SQL statements representing checks from an ASCII file. It executes these SQL statements by calling a Visual C++ application with the SQL statement as a parameter and printing the results to an output file. The results of the checker are lists of problem descriptions followed by the elements that have this problem. The results are plain ASCII.

The analyser creates and fills the database (using database creator, database filler and the database checker). After which it reads the SQL statements representing the metrics from an ASCII file. It executes these SQL statements by calling a Visual C++ application with the SQL statement as a parameter and printing the results to an output file. The results of the analyser are again comma separated ASCII.

The statistic calculator reads the results of the analyser and calculates the statistics; these statistics are printed to an output file.

The statistic filter reads the results of the analyser and the statistics of the statistic calculator and prints only the elements of the result that have outlying values. Again these values are printed to an output file.

Saat controls the main program flow. Saat is a .bat-file that sequentially calls the analyser and the filter component.

For more information on the implementation of the components of the Software Architecture Analysis Tool see appendix Code. This appendix covers the implementation in more depth and contains all the code written in order to implement the tool.
9 The Results

During this project the Software Architecture Analysis Tool was not only implemented. The analysis methods have been tested on several real architectures in order to verify the metrics and techniques. This chapter discusses the results of these analyses.

The first impression of the analysis are promising, even though some of the architects that provided the architectures are sceptic and not convinced that the metrics provide information on the quality of the architecture. It seems that the tool is able to extract critical components in the software architecture. Reasoning about these critical components and searching for an explanation why this component is critical helps the architect improving his design.

Here are some examples of questions that help the architect improving his design:

- "Why does this component need so many services?"
- "Why is this component used for so many use cases?"
- "Why do so many other components depend on this component?"
- "Why is this scenario so complex?"
- ...

There is a problem. An architecture is described by means of diagrams. These diagrams are the input for the analysis tool. This means that the tool analyses architectures by means of analysing architecture descriptions. This is the only possible approach but has the limitation that the results depend on choices of the architect. For example: The architect illustrates some of the important use cases by means of scenarios. In these scenarios he illustrates how the components that he thinks important work together. The results of the analysis depend on the choices of the architect regarding the description of the architecture.
9.1 MHP Architecture

The MHP architecture describes a box used to extend the functionality of a television. The television is extended with functionality necessary for interactive television. The box is being developed at Philips Eindhoven.

The architecture is quite large.

<table>
<thead>
<tr>
<th>Number of use cases</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components</td>
<td>163</td>
</tr>
<tr>
<td>Number of scenarios</td>
<td>39</td>
</tr>
<tr>
<td>Number of services</td>
<td>1018</td>
</tr>
<tr>
<td>Number of states (sum over all components)</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

The tool was used to calculate the metrics and filter the results for outlying values. The next table shows the results of these analyses. This table shows the number of elements that are indicated as special / problem elements, the percentage of elements selected and the number of elements that are accepted / rejected as problem elements by one of the architects.

<table>
<thead>
<tr>
<th>Metric</th>
<th># Outlying values</th>
<th>% of outlying values</th>
<th># False Accepts</th>
<th># False Rejects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components per use case</td>
<td>1</td>
<td>1/18</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Number of use cases per component</td>
<td>1</td>
<td>1/163</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Number of called services per component</td>
<td>3</td>
<td>3/163</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Number of service-calls by component</td>
<td>2</td>
<td>2/163</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Number of services per component</td>
<td>12</td>
<td>12/163</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Complexity of services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Lack of cohesion between services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Depth of scenario</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
</tbody>
</table>
9.2 Elevator Architecture

The architecture describes an elevator system. The system has been designed with much attention for extendibility and portability. The system is designed at Eindhoven University of Technologies by Onno v. Roosmalen.

The system is quite small.

<table>
<thead>
<tr>
<th>Number of use cases</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components</td>
<td>19</td>
</tr>
<tr>
<td>Number of scenarios</td>
<td>1</td>
</tr>
<tr>
<td>Number of services</td>
<td>42</td>
</tr>
<tr>
<td>Number of states (sum over all components)</td>
<td>21</td>
</tr>
</tbody>
</table>

The tool was used to calculate the metrics and filter the results for outlying values. This analysis produced almost no indication of bottlenecks. There was only one component marked as a problem component because of the high number of services it provides. The architect was of the opinion that this was not a problem.

<table>
<thead>
<tr>
<th>Metric</th>
<th># Outlying values</th>
<th>% of outlying values</th>
<th># False Accepts</th>
<th># False Rejects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components per use case</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of use cases per component</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of called services per component</td>
<td>0</td>
<td>0/19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of service-calls by component</td>
<td>0</td>
<td>0/19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of services per component</td>
<td>1</td>
<td>1/19</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Complexity of services</td>
<td>0</td>
<td>0/19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Coupling</td>
<td>0</td>
<td>0/19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inverse coupling</td>
<td>0</td>
<td>0/19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lack of cohesion between services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Depth of scenario</td>
<td>0</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because of the small number of outlying values we studied the results of the metrics more carefully. I found two components that had a high values (but less than 2 times the standard deviation) on several metrics (Number of called services of component, Number of service-calls by component, Number of services of component, Complexity of services and Inverse coupling). One component is rejected and one is accepted as a problem element by the architect.
9.3 Management Application

This architecture describes a management system. The architecture has been provided by Ronald Pulleman.

Size of the architecture:

<table>
<thead>
<tr>
<th>Metric</th>
<th>n.a.</th>
<th>110</th>
<th>22</th>
<th>n.a.</th>
<th>423</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of use cases</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of components</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of scenarios</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of services</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of states (sum over all components)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The tool was used to calculate the metrics and filter the results for outlying values. The next table shows the results of these analyses. This table shows the number of elements that are indicated as special / problem elements, the percentage of elements selected and the number of elements that are accepted / rejected as problem elements by one of the architects.

<table>
<thead>
<tr>
<th>Metric</th>
<th># Outlying values</th>
<th>% of outlying values</th>
<th># False Accepts</th>
<th># False Rejects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components per use case</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of use cases per component</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of called services per component</td>
<td>2</td>
<td>2/110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of service-calls by component</td>
<td>2</td>
<td>2/110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of services per component</td>
<td>5</td>
<td>5/110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complexity of services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Coupling</td>
<td>2</td>
<td>2/110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inverse coupling</td>
<td>1</td>
<td>1/110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lack of cohesion between services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Depth of scenario</td>
<td>0</td>
<td>0/110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width of scenario</td>
<td>3</td>
<td>3/110</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9.4 Expresso Architecture

This architecture describes an offering system. The architecture has been provided by Ronald Pulleman.

The architecture is quite large.

<table>
<thead>
<tr>
<th>Number of use cases</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components</td>
<td>2653</td>
</tr>
<tr>
<td>Number of scenarios</td>
<td>15</td>
</tr>
<tr>
<td>Number of services</td>
<td>4146</td>
</tr>
<tr>
<td>Number of states (sum over all components)</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

The tool was used to calculate the metrics and filter the results for outlying values. The next table shows the results of these analyses. This table shows the number of elements that are indicated as special / problem elements, the percentage of elements selected and the number of elements that are accepted / rejected as problem elements by one of the architects.

<table>
<thead>
<tr>
<th>Metric</th>
<th># Outlying values</th>
<th>% of outlying values</th>
<th># False Accepts</th>
<th># False Rejects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components per use case</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of use cases per component</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of called services per component</td>
<td>1</td>
<td>1/2653</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of service-calls by component</td>
<td>1</td>
<td>1/2653</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Number of services per component</td>
<td>15</td>
<td>15/2653</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Complexity of services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Coupling</td>
<td>2</td>
<td>2/2653</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Inverse coupling</td>
<td>2</td>
<td>2/2653</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Lack of cohesion between services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Depth of scenario</td>
<td>1</td>
<td>1/15</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Width of scenario</td>
<td>1</td>
<td>1/15</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
</tbody>
</table>
9.5 Order–System Architecture

This architecture is an example of the Rational Rose modelling tool, therefore it is not possible to let the architect verify the results. For this case I accepted / rejected the problem elements myself.

<table>
<thead>
<tr>
<th>Number of use cases</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components</td>
<td>478</td>
</tr>
<tr>
<td>Number of scenarios</td>
<td>2</td>
</tr>
<tr>
<td>Number of services</td>
<td>95</td>
</tr>
<tr>
<td>Number of states (sum over all components)</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

The tool was used to calculate the metrics and filter the results for outlying values. This analysis produced almost no indication of bottlenecks. There were only two components marked as a problem component. One because of the high number of services it provides and one because of the number of components depending on it (inverse coupling). With the component marked based on the number of services it provides I agree. The second marking (based on inverse coupling) is not a problem element, the only reason it has been marked as a problem element is that all the other components have just one component depending on them.

<table>
<thead>
<tr>
<th>Metric</th>
<th># Outlying values</th>
<th>% of outlying values</th>
<th># False Accepts</th>
<th># False Rejects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components per use case</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of use cases per component</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of called services per component</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of service-calls by component</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Number of services per component</td>
<td>1</td>
<td>1/478</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Complexity of services</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
<tr>
<td>Coupling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Inverse coupling</td>
<td>1</td>
<td>1/478</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Lack of cohesion between services</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Depth of scenario</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Width of component in scenario</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Because of the small number of outlying values I studied the results of the metrics more carefully. I have found one component that has a high score on several metrics (number of service calls, number of called services, width of component in scenario and coupling). It’s my opinion that this component is a critical component.
10 Future Work

The Software Architecture Analysis Tool is a start in automated software architecture analysis. A lot of improvements can still be made. This appendix presents a list of recommended improvements.

- **Packages**: Currently package information in the software architecture is ignored. Packages give lots of information on the organisation of a software architecture. With package information new metrics are possible and package information tells a lot about the quality of software architectures.
- **Installation program**: The Software Architecture Analysis Tool lacks an installation program. The tool requires the installation of mySQL-server and of ActivePerl. These programs must be installed correctly and the files of the tool must be copied before the tool can be used. Currently an installation manual exists, however the installation procedure can still be simplified and automated.
- **Presentation metrics**: Currently the tool shows the results of the metrics by means of tables. In order to interpret the results more quickly graphs can be used to present the results.
- **Conclusions based on metrics**: The Software Architecture Analysis Tool computes metrics. However an architect is not interested in the scores, he is interested in what kind of improvements he can make on his design, he is interested in whether his design is good or bad, he wants to know what the bottlenecks are, etc...
- **Improvement suggestions**: The goal of software architecture analysis is to improve a design. A tool that indicates problems is good, a tool that solves problems is better.
- **Normalisation**: In order to be able to compare the results of the analysis of several architectures the metrics should be normalised. This makes it possible to compare elements of different architectures and maybe even make statements on what is the best architecture.
- **More / better metrics**: Currently the Software Architecture Analysis Tool is not only used for the analysis of architectures. Besides for the analysis of architectures the tool can be used to study the several possible metrics for software architectures (which are useful and which useless).
- **Distinguish analysis of architecture and analysis of diagram**: There is a difference between the diagrams describing an architecture and the architecture itself. Diagrams can be too detailed, the wrong abstraction level can be used for a diagram which does not mean that the architecture is bad. Currently the tool does not distinguish this difference.
- **ODBC**: The tool uses a mySQL-server. By using ODBC drivers the implementation of the tool can be made independent on mySQL. This means that the mySQL-server can be replaced by any other SQL-server supporting ODBC.
11 Conclusions and Recommendations

Analysing software architectures is a complicated task. Several methods have been designed to evaluate software architectures. SAAM and ATAM are the most well known. The most important thing those methods have in common is that they use experienced architects for the evaluation of a design.

The Software Architecture Analysis Tool evaluates a design without the help of an architect. However the results should be interpreted by an architect and the architect can take advantage of the information given by the tool. The Software Architecture Analysis Tool does not tell whether an architecture is good or bad, but it helps the architect in improving his design by indicating possible problem elements.

The Software Architecture Analysis Tool can be used to test / verify evaluation algorithms. The tool is designed in such a way that evaluation algorithms can easily be added, removed and changed. This is very useful because the algorithms / metrics currently implemented need to be improved / refined in order to increase the added value of the analysis tool.

The algorithms currently implemented are an arbitrary selection. Several other algorithms are available or can be designed. This means that the Software Architecture Analysis Tool is not finished. New algorithms can be added and the existing ones can be polished. I recommend that the existing metrics are verified on a larger number of test–cases and that new metrics are added.

It is my belief that the Software Architecture Analysis Tool can be of great help for a software architect. However once again it is not finished and therefore I would like to point to the chapter “Future work”.

That the tool is not finished does not mean that it does not already provide useful information. Several tests showed that after a short but critical view on the results of the analysis tool a very good guess on what are problem elements can be made. Reasoning about these problem elements and finding explanations for the high scores helps improving the software architecture.
Literature

Books


Papers

[B] Staan op schouders van reuzen, Bjorn Bon
[D] Metrification of a software architecture definition; F.W. Greuter
[F] The Koala Component Model for Consumer Electronics Software; Rob van Ommering, Frank van der Linden, Jeff Kramer and Jeff Magee (2000)
[H] Architectural Blueprints --- The 4+1 View Model of Software Architecture; Philippe Kruchten (1995)
[L] Scripting Coordination Styles; Franz Achermann, Stefan Kneubuehl and Oscar Nierstrasz (2000)
Appendices

A  Code

File: extractUseCases.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Use cases in the xmi file (INPUT) are written to an output file (OUTPUT). There fore the following syntax is used: ID, NAME.
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl

open(IN,  "<SARGV[0]") unless SARGV[0];
open(OUT,  ">SARGV[1]");
open(ERR,  ">>SARGV[2]");

@input = <IN>

@tags = split(/
", @input);

foreach $tag (@tags)
  { 
    if($tag =~ /\s*<\s*UML::UseCase\s*\>/)
      {
        if($tag =~ /\s*id\s*\=(\s*\{.*\})\s*\>/)
          { 
            next if $2 eq "";
            print OUT "$2, \n";
          }
        else
          { 
            print ERR "extractUseCases: cannot find name or id\n";
          }
      }
  }

```

Version: 1.10
Date: 22 March 2002
Software Architecture Analysis Tool

File: extractScenarios.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is a filename of a model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Components (classes) in the xmi file (INPUT) are written to an output file (OUTPUT). Therefore, the following syntax is used: ID, NAME.
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl

# -- open files
open(INPUT, "<$ARGV[0]"/linux
open(OUTPUT, ">$ARGV[1]"/ubuntu
open(ERROR, ">$ARGV[2]"/windows

# -- read input in array
@lines = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split('/',join(' ',@lines));

# -- check all tags whether it's a scenario tag
foreach $tag (@tags)
{
    # -- does tag contain string "UML:Interaction"? (match case insensitive)
    if($tag =~ /<UML::Interaction>/i)
    {
        # -- find attributes xml.id ($1) and name ($2) of the tag
        if($tag =~ /xml.id="s*"\s*\{[\w\\\w\(\)\{\}\[\]\_\-\:\.]\}s*\}([\w\\\w\(\)\{\}\[\]\_\-\:\.]\}s*/i)
        {
            # -- output scenario information (id,name)
            print OUTPUT "$1,$2\n";
        }
    } else
    {
        # -- output error information
        print ERROR 'extractScenarios: cannot find id or name\n';
        print ERROR "$tag\n";
    }
}
```
File: extractComponents.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Use cases in the xmi file (INPUT) are written to an output file (OUTPUT). There fore the following syntax is used: ID, NAME.
- Errors are written to an output file (ERROR).

```
#!/usr/bin/perl

# -- open files
open(INPUT,"<$ARGV[0]" );
open(OUTPUT,">$ARGV[1]" );
open(ERROR,">$ARGV[2]" );

# -- read input in array
@lines = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split(/
/,join(
, @lines));

# -- check all tags whether it's a scenario tag
foreach $tag (@tags)
{
  # -- does tag contain string "UML:Class" or "UML:Interface" ?
  # [match case insensitive]
  if($tag =~ /UML:Class /gi || $tag =~ /UML:Interface /gi)
  {
    # -- find attributes xml.id ($1) and name ($2) of the tag
    if($tag =~ /xml:id\"s*=\"s*([\w\s\{\}\(\)\|\+\-\=\*\+\-\+=\-\-\]+\//gi)
      {
        # -- output component information
        print OUTPUT "$1,$2\n";
      }
    else
      {
        # -- output error information
        print ERROR "extractComponent: cannot find id or name\n";
        print ERROR "$tag\n";
      }
  }
}
```
File: extractServices.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Services (methods of classes) in the xmi file (INPUT) are written to an output file (OUTPUT). Therefore the following syntax is used: ID, NAME.
- Errors are written to an output file (ERROR).

```perl
#!c:/perl/bin/perl;

# -- open files
open(INPUT, "<$ARGV[0]">);
open(OUTPUT, ">$ARGV[1]">);
open(ERROR, ">$ARGV[2]">);

# -- read input in array
@lines = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split(/\>/, join('', @lines));

# -- check all tags whether it's a service tag
foreach $tag (@tags) {
    # -- does tag contain string "UML:Operation" ? (match case insensitive)
    if($tag =~ /UML:Operation/gi) {
        # -- find attributes xml.id ($1) and name ($2) of the tag
        if($tag =~ /xml.id\s*=\s+['"]\w+(\s+['"]\w+)+['"]\s*}$/gi) {
            # -- output services information
            print OUTPUT "$1,$2\n";
        } else {
            # -- output error information
            print ERROR "extractServices: cannot find name or id\n";
            print ERROR "$tag\n\n";
        }
    }
}```
File: extractStates.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- States in the xmi file (INPUT) are written to an output file (OUTPUT). Therefore the following syntax is used: ID, NAME.
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl

# -- open files
open(INPUT, "<$ARGV[0]>");
open(OUTPUT,">>$ARGV[1]>");
open(ERROR,">>$ARGV[2]>");

# -- read input in array
@lines = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split('/>', join('', @lines));

# -- check all tags whether it's a state tag
foreach $tag (@tags)
{
    # -- does tag contain string "UML: SimpleState " ? (match case insensitive)
    if($tag =~ /UML:SimpleState /gi)
    {
        # -- find attributes xml:id ($1) and name ($2) of the tag
        if($tag =~ /xml:id="s*=\"[^\"]*\"/)
        {
            # -- output state information
            print OUTPUT "$1,$2\n";
        }
        else
        {
            # -- output error information
            print ERROR "extractStates: cannot find name or id\n";
            print ERROR "$tag\n";
        }
    }
}
```
File: extractUS.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT,OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose
  UniSysRoseXMLTools.

Post-condition:
- Use case - Scenario (has) relations in the xmi file (INPUT) are written to an output
  file (OUTPUT). There fore the following syntax is used: ID_U, ID_S.
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl;

# -- open files
open(INPUT, "<ARGV[0]");
open(OUTPUT, "<ARGV[1]");
open(ERROR, "<ARGV[2]");

# -- read input to array
@lines = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split("/", join('', @lines));

# -- search all tags sequentially
$UseCaseId = "";
foreach @tag (
  # -- first find the tag containing the string "<UML:UseCase " (match case insensitive)
  if($tag =~ /<UML:UseCase /gi)
  {
    # -- find xmi.id attribute ($1) of the tag
    if($tag =~ /xmi.id="s*'<(\"\"|\\\")\/_;\;\;\;\-_+'/gi)
    {
      # -- store xmi.id attribute
      $UseCaseId = $1;
    } else {
      # -- output error information
      print ERROR "extractUS: Cannot use case find id\n";
      print ERROR "$tag\n";
    }
  }

  # -- then find tag containing string "<UML:Interaction " (match case insensitive)
  if($tag =~ /<UML:Interaction /gi)
  {
    # -- find attribute xmi.id ($1)
    if($tag =~ /xmi.id="s*'<(\"\"|\\\")\/_;\;\;\;\-_+'/gi)
    {
      # -- is it before the close tag of UML:UseCase ?
      if($UseCaseId)
      {
        # -- output U-S relation
        print OUTPUT "$UseCaseId,$1\n";
      } else {
```

Page: 62 / 105
Version: 1.10
Date: 22 March 2002
{  
   # -- output error information  
   print ERROR "ExtractUS: No Use-Case id\n";  
   print ERROR "$tag\n\n"  
}  
else  
{  
   # -- output error information  
   print ERROR "ExtractUS: cannot find scenario id\n";  
   print ERROR "$tag\n\n";  
}  

# -- finally find close tag.  
if($tag == \/<\!/UML:UseCase$/gi)  
{  
   $UseCaseId = "";  
}  
}
File: extractCM.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Component - method (has) relations in the xmi file (INPUT) are written to an output file (OUTPUT). Therefore the following syntax is used: ID_C, ID_M.
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl

# -- open files
open(INPUT, "<ARGV[0]'");
open(OUTPUT, "<ARGV[1]'");
open(ERROR, "<ARGV[2]'");

# -- read input in an array
@input = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split('/', join('', @input));

# -- search tags sequentially
$ComponentId = "";
foreach $tag (@tags) {
    # -- first find tag containing string "<UML:Class" or "<UML:Interface"
    if($tag =~ /<UML:Class/i || $tag =~ /<UML:Interface/i)
    {
        # -- find tag attribute xmi:id ($1)
        if($tag =~ /xmi:id="xmi:id='([^\s\n\|\s]*\s*[^\s\n\|\s]*)'/i)
        {
            # -- store attribute xmi:id
            $ComponentId = $1;
        }
        else
        {
            # -- output error information
            print ERROR "extractCM: cannot find class id
"
            print ERROR "$tag
"
        }
    }

    # -- then find tag containing string "<UML:Operation" (match case insensitive)
    if($tag =~ /<UML:Operation/i)
    {
        # -- find tag attribute xmi:id ($1)
        if($tag =~ /xmi:id="xmi:id='([^\s\n\|\s]*\s*[^\s\n\|\s]*)'/i)
        {
            # -- is tag within scope of UML:Class tag ?
            if($ComponentId)
            {
                # -- output C-M relation
                print OUTPUT "$ComponentId,$1
"
            }
        }
    }
}
```
else
{
    # -- output error information
    print ERROR "extractCM: no component Id\n";
    print ERROR "$tag\n\n";
}
}
else
{
    # -- output error information
    print ERROR "extractCM: cannot find Operation id\n";
    print ERROR "$tag\n\n";
}

# -- last find the close tag
if($tag == /</UML:Class$/gi || $tag == /</UML:Interface$/gi)
{
    $ComponentId = "";
}
}
File: extractSM.pl  
Arguments: <INPUT, OUTPUT, ERROR>  
Language: Perl  
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Service calls of the scenarios in the xmi file (INPUT) are written to an output file (OUTPUT). Therefore the following syntax is used: ID, ID_S, ID_C1, ID_C2, ID_M2, ID_PRED.
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl

# -- open files
open(INPUT, "<$ARGV[0]" );
open(OUTPUT, ">$ARGV[1]" );
open(ERROR, ">$ARGV[2]" );

# -- read input in array
$lines = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split(';', join(',', @lines));

# -- search all tags sequentially and make hashtable containing
# -- CallActionId's and their corresponding Foundation.Core.Operations
$CallActionId = "";
$CallActionList = ();
foreach $tag (@tags)
{
    # -- find tag containing string "<UML:CallAction" (match case insensitive)
    if($tag =~ /<UML:CallAction/i)
    {
        # -- find tag attribute xmi.id ($1)
        if($tag =~ /xmi.id=".*"/)
        {
            # -- store attribute xmi.id
            $CallActionId = $1;
        }
        else
        {
            # -- output error information
            print ERROR "extractSM: cannot find callAction id
";
            print ERROR "$tag
";
        }
    }

    # -- then find tag containing string "<Foundation.Core.Operation" (match case insensitive)
    if($tag =~ /<Foundation.Core.Operation/i)
    {
        # -- find tag attribute xmi.idref ($1)
        if($tag =~ /xmi.idref=".*"/)
        {
            # -- store in hashtable
            $CallActionList{$CallActionId} = $1;
        }
        else
        {
            # -- output error information
            print ERROR "extractSM: cannot find callAction id
";
            print ERROR "$tag
";
        }
    }
}```
Software Architecture Analysis Tool

{  
    # -- output error information
    print ERROR "extractSM: cannot find core operation idref\n";
    print ERROR "$tag\n";
}

# -- finally find close tag
if($tag =~ /<\/UML:CallAction$/) {
    $CallActionId = "";
}

# search all tags sequentially and puzzle with id's, idrefs, etc, etc....... 
$base = 0;
$type = 0;
$sender = "";
$receiver = "";
$predecessor = "";
$action = "";
$ClassifierRoleId = "";
$CollaborationId = "";
$InteractionId = "";
$MessageId = "";
%ClassifierList = ();
foreach $tag (@tags) {
    # -- find collaboration
    if($tag =~ /<UML:Collaboration /gi) {
        if($tag =~ /xml.id\s*=\s*"\{[\w\s\(\)\{\}\_.;\-]+\}"/gi) {
            $CollaborationId = $1;
        } else {
            print ERROR "extractSM: cannot find collaboration id\n";
            print ERROR "$tag\n";
        }
    }
    # -- find classifierRole
    if($tag =~ /<UML:ClassifierRole /gi) {
        if($tag =~ /xml.id\s*=\s*"\{[\w\s\(\)\{\}\_.;\-]+\}"/gi) {
            $ClassifierRoleId = $1;
        } else {
            print ERROR "extractSM: cannot find classifierrole id\n";
            print ERROR "$tag\n";
        }
    }
    # -- find classifierRoleBase
    if($tag =~ /<UML:ClassifierRole.base/gi) {
        $base = 1;
    }
    # -- find foundation core classifier
    if($tag =~ /<Foundation.Core.Classifier/gi) {
        if($tag =~ /xml.idref\s*=\s*"\{[\w\s\(\)\{\}\_.;\-]+\}"/gi) {
            if($ClassifierRoleId) {
                $ClassifierList{$ClassifierRoleId} = $1;
            }
        } else {
            
        }
    }
}
print ERROR "extractSM: cannot find core classifier id\n";
print ERROR "$tag\n";
}

# -- find interaction (messages are part of an interaction)
if($tag =~ /<UML:Interaction /gi)
{
    if($tag =~ /xmi.id='s*'=\w+\s+'([^\w\s\(\)\{\}\[\]|/\.;:]*-)+')/gi)
    {
        $InteractionId = $1;
    }
    else
    {
        print ERROR "extractSM: cannot find interaction id\n";
        print ERROR "$tag\n";
    }

# -- find message (service call is a message)
if($tag =~ /<UML:Message /gi)
{
    if($tag =~ /xmi.id='s*'=\w+\s+'([^\w\s\(\)\{\}\[\]|/\.;:]*-)+')/gi)
    {
        $MessageId = $1;
    }
    else
    {
        print ERROR "extractSM: cannot find message id\n";
        print ERROR "$tag\n";
    }

# -- determine type of message
if($tag =~ /<UML:Message.sender/gi)
{
    $type = 0;
}

# -- find sender / receiver depending on message type
if($tag =~ /<Behavioral_Elements.Collaborations.ClassifierRole/gi)
{
    if($tag =~ /xmi.idref='s*'=\w+\s+'([^\w\s\(\)\{\}\[\]|/\.;:]*-)+')/gi)
    {
        if($type == 0)
        {
            $sender = $1;
        }
        if($type == 1)
        {
            $receiver = $1;
        }
    }
    else
    {
        print ERROR "extractSM: cannot find classifierrole id\n";
        print ERROR "$tag\n";
    }

# -- determine type of message
if($tag =~ /<UML:Message.predecessor$/gi)
{
    $type = 3;
}

# -- find predecessor
if($tag =~ /Behavioral_Elements.Collaborations.Message /gi)
{
    if($type == 3)
    {
        if($tag =~ /xmi.idref='s*'=\w+\s+'([^\w\s\(\)\{\}\[\]|/\.;:]*-)+')/gi)
        {
            $predecessor = $1;
        }
    }
}
{
    print ERROR "extractSM: cannot find message idref\n";
    print ERROR "$tag\n\n";
}
}

# -- determine type of message
if($tag =~ /<UML:Message.receiver\/>i) {
    $type = 1;
}

# -- determine type of message
if($tag =~ /<UML:Message.action\/>i) {
    $type = 4;
}

# -- find service of the call
if($tag =~ /<Behavioral_Elements.Common_Behavior.Action /i) {
    if($type == 4) {
        if($tag =~ /xmi.idref\$=\$\*\((\w\s{1,1}\{1,1}\w+;\.;\;\+\}\)/i) {
            $action = $1;
        } else {
            print ERROR "extractSM: cannot find action idref\n";
            print ERROR "$tag\n\n";
        }
    }
}

# -- find close tag (now we can output service-call)
if($tag =~ /</\UML:Message$/i) {
    if($InteractionId && $MessageId) {
        print OUTPUT
        "$MessageId,$InteractionId,$ClassifierList($sender),
        $ClassifierList($receiver),$CallActionList($action),
        $precedessor\n";
        if(!($ClassifierList($sender) && $ClassifierList($receiver) &&
            $CallActionList($action))) {
            print ERROR "extractSM: no sender, no receiver or no
callaction\n";
            print ERROR "$MessageId:$MessageId\n\n";
        } else {
            print ERROR "extractSM: no InteractionId or no MessageId\n\n";
        }
    } else {
        $sender = "";
        $receiver = "";
        $precedessor = "";
        $action = "";
        $MessageId = "";
    }

    # -- find close tag of UML:Interaction
    if($tag =~ /</\UML:Interaction$/i) {
        $InteractionId = "";
    }

    # -- find close tag of UML:ClassifierRole.base
    if($tag =~ /</\UML:ClassifierRole.base$/i) {
        $base = 0;
    }
}
# -- find close tag of UML:ClassifierRole
if($tag =~ /<\/UML:ClassifierRole$/gi) {
    $ClassifierRoleId = "";
}

# -- find close tag of UML:Collaborations
if($tag =~ /<\/UML:Collaboration$/gi) {
    $CollaborationId = "";
    $ClassifierRoleId = "";
    $ClassifierList = ();
}
File: extractCT.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Component – State (has) relations in the xmi file (INPUT) are written to an output file (OUTPUT). Therefore the following syntax is used: ID_C, ID_T.
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl

# -- open files
open(INPUT, "<$ARGV[0]"),
open(OUTPUT, ">$ARGV[1]"),
open(ERROR, ">$ARGV[2]"),

# -- read input in array
@lines = <INPUT>;

# -- join elements of lines then split at tag boundary
@tags = split(/\>/,.join(",", @lines));

# -- search all tags sequentially
$StateMachineId = ""
$ComponentId = ""
foreach $tag (@tags) {
    # -- first find tag containing string "<UML:StateMachine" (match case insensitive)
    if ($tag =~ /<UML:StateMachine /gi)
        {
            # -- find tag attribute xml:id ($1)
            if ($tag =~ /xml:id="[a-zA-Z0-9_\-/]+">/gi)
                {
                    # -- store attribute xml:id
                    $StateMachineId = $1;
                    $ComponentId = ""
                }
            else
                {
                    # -- print error information
                    print ERROR "extractCT: cannot find id or name\n";
                    print ERROR "$tag\n"
                }
        }
    # -- then find the tag with string "Foundation.Core.ModelElement"
    # (this is the component)
    if ($tag =~ /Foundation.Core.ModelElement/gi)
        {
            # -- find the tag attribute xml:idref ($1)
            if ($tag =~ /xml:idref="[a-zA-Z0-9_\-/]+">/gi)
                {
                    if ($ComponentId)
                        {
                            # -- store attribute xml:idref
                            $ComponentId = $1;
                        }
                    else
                        {
                            # -- store attribute xml:idref
                            $ComponentId = $1;
                        }
                }
        }
}
```

Version: 1.10
Date: 22 March 2002
} else {
    # -- output error information
    print ERROR "extractCT: cannot find idref\n";
    print ERROR "$tag\n\n";
}

# -- then find the tag with string "<UML:SimpleState" (match case insensitive)
if($tag == /<UML:SimpleState/gi) {
    # -- find tag attribute xml.id ($1)
    if($tag == /xml.id='[^\w\s\(\)\{\}\[\]]+(\.|\d+)\n"/gi) {
        if($StateMachineId & $ComponentId) {
            # -- output C-T relation
            print OUTPUT "$ComponentId,$1\n";
        } else {
            # -- output error information
            print ERROR "extractCT: no state machine id or no component id\n";
            print ERROR "$tag\n\n";
        }
    } else {
        # -- output error information
        print ERROR "extractCT: cannot find state id\n";
        print ERROR "$tag\n\n";
    }
}

# -- finally find close tag and start over
if($tag == /</UML:StateMachine$/gi) {
    $StateMachineId = "";
    $ComponentId = "";
}
File: extractTT.pl
Arguments: <INPUT, OUTPUT, ERROR>
Language: Perl
Author: Johan Muskens

Implementation for component: Parser

Pre-condition:
- INPUT, OUTPUT and ERROR are valid filenames.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- State – State (can change to) relations in the xmi file (INPUT) are written to an output file (OUTPUT). Therefore the following syntax is used: ID_T1, ID_T2
  (FROM STATE, TO STATE).
- Errors are written to an output file (ERROR).

```perl
#!/usr/bin/perl

# -- open files
open(INPUT, "<$ARGV[0]");
open(OUTPUT, ">$ARGV[1]");
open(ERROR, ">>$ARGV[2]");

# -- read input in array
@lines = <INPUT>;

# -- join elements of lines and then split at tag boundary
@tags = split(/>/, join(',', @lines));

# -- search all tags sequentially
$TransitionId = "";
$FromStateId = "";
$ToStateId = "";
$StateType = 0;
foreach $tag (@tags) {
  # -- first find tag with string "<UML:Transition" (match case insensitive)
  if($tag =~ /<UML:Transition/gi) {
    # -- find tag attribute xml.id ($1)
    if($tag =~ /xml.id="s*(\w*:\\s\w*("\w*","\w*")\w*\)/gi) {
      # -- store attribute xml.id
      $transitionId = $1;
    } else {
      # -- output error information
      print ERROR "extractTT: can not find Transition id\n";
      print ERROR "$tag\n";
    }
  }

  # -- determine whether next StateVertex is from or to state.
  if($tag =~ /<UML:Transition.source/gi) {
    $stateType = 0;
  }

  # -- determine whether next StateVertex is from or to state.
  if($tag =~ /<UML:Transition.target/gi) {
```

Version: 1.10
Date: 22 March 2002
{  $StateType = 1;
}

# -- find StateVertex (tag containing string  
# "Behavioral_Elements.State_Machines.StateVertex")  
if($tag =~ /Behavioral_Elements.State_Machines.StateVertex/gi)
{
  # -- find tag attribute xml:idref ($)  
  if($tag =~ /xml:idref\"s*'"(\W\s\(\|\|\)\|\(\)\;\;\;\;\-\)+)'/gi)
  {
    # -- store state depending on statetype  
    if($StateType == 0)
    {
      $FromStateId = $1;
    }
    else
    {
      $ToStateId = $1;
    }
  }
  else
  {
    # -- output error information  
    print ERROR "extractTT: cannot find state idref\n";
    print ERROR "$tag\n";
  }
}

# -- finally find close tag (now we can output transition information)  
if($tag =~ /</UML:Transition$/gi)
{
  if($FromStateId && $ToStateId)
  {
    print OUTPUT "$FromStateId,$ToStateId\n";
  }
  else
  {
    print ERROR "extractTT: No FromState or no ToState found\n";
    print ERROR "Transition id:$TransitionId\n";
  }
  
  $TransitionId = "";
  $FromStateId = "";
  $ToStateId = "";
  $StateType = 0;
}
Implementation for component: Database Creator

Pre-condition:
- HOST, NAME, PASSWD give access to MySQL database server containing Architectures database.
- libMySQL.dll is in dll-path

Post-condition:
- New database is created with the Tables U, S, C, M, T, Us, Sm, Cm, Ct and Tt
- Name of the created database is written to std out.

```cpp
// MySQLCreateDb.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include "MySQLCreateDb.h"

#include <iostream>
#include <iomanip>
#include <mysql++>

#ifndef __DEBUG
#define DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


// The one and only application object
CWinApp theApp;

using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    int nRetCode = 0;

    // initialize MFC and print and error on failure
    if (!AFXWinInit::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
    {
        // TODO: change error code to suit your needs
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
    }
    else
    {
        try {
            // fetch program arguments host, user and password
            string host = "";
            string user = "";
            string passwd = "";
            if(argc < 4)
            {
                return -1;
            }
```

Version: 1.10
Date: 22 March 2002
else
{
    host = argv[1];
    user = argv[2];
    passwd = argv[3];
}

// Make connection with SQL server
Connection
con("Architectures", host.c_str(), user.c_str(), passwd.c_str());

// Find new database name (from Table A in database
// Architectures)
Query query = con.query();
query << "SELECT * FROM A;";
Result res = query.store();
int s = res.size();

CString name;
char tmp[10];
itoa(s, tmp, 10);
name = "Architectuur";
name += tmp;

// Add to be created database to table A of Architectures
query << "INSERT INTO A (ID,NAME) VALUES (" << s << "," "Architectuur" << s << ");";
res = query.store();

// Create new database
con.create_db((LPCTSTR)name);
con.select_db((LPCTSTR)name);

// Create New Tables
query = con.query();
query << "CREATE TABLE U (ID CHAR(10),NAME CHAR(50));";
res = query.store();
query << "CREATE TABLE S (ID CHAR(10),NAME CHAR(50));";
res = query.store();
query << "CREATE TABLE C (ID CHAR(10),NAME CHAR(50));";
res = query.store();
query << "CREATE TABLE M (ID CHAR(10),NAME CHAR(50));";
res = query.store();
query << "CREATE TABLE T (ID CHAR(10),NAME CHAR(50));";
res = query.store();
query << "CREATE TABLE US (ID_U CHAR(10),ID_S CHAR(10));";
res = query.store();
query << "CREATE TABLE CM (ID_C CHAR(10),ID_M CHAR(10));";
res = query.store();
query << "CREATE TABLE SM (ID CHAR(10),ID_S CHAR(10),ID_C CHAR(10),ID_C1 CHAR(10),ID_C2 CHAR(10),ID_M2 CHAR(10),ID_FRED CHAR(10));";
res = query.store();
query << "CREATE TABLE CT (ID_C CHAR(10),ID_T CHAR(10));";
res = query.store();
query << "CREATE TABLE TT (ID_T1 CHAR(10),ID_T2 CHAR(10));";
res = query.store();

// Output created database name
cout << (LPCTSTR)name;
}
catch (BadQuery er)
{
    cerr << "Error: " << er.error << endl;
nRetCode = -1;
}
catch (BadConversion er)
{  
cerr << "Error: Tried to convert \" << er.data << "\" to a \" <<  
   er.type_name << ")\"." << endl;  
nRetCode = -1;  
}

return nRetCode;
File: fillU.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Use cases in the xmi file (INPUT_FILE) are stored in de database written in DATABASE_NAME_FILE.

```perl
#!/usr/bin/perl

# -- extract Usecases from xmi-file and write to text-file using format ID,NAME,
# "\Parser\extractUseCases.pl $ARGV[1] ..\Parser\Output\U.txt
# ..\Parser\Output\Parser.log"

# -- find out which database to store architecture information
open(ARCH_FILE,$ARGV[0]);
$architecture = <ARCH_FILE>;

# -- open file containing extracted information
open(U_FILE,"..\Parser\Output\U.txt");

# -- each line contains a record for table U
@lines = <U_FILE>;
foreach $line (@lines)
{
    # -- empty line?
    if($line)
    {
        # -- split at , (remaining are two fields, first is id, second is name
        @fields = split ",",$line,2;
        $id = $fields[0];
        $name = $fields[1];
        chomp($id);
        chomp($name);

        # -- insert into table U using MySQLInsert
        $query = "INSERT INTO U (ID,NAME) VALUES('$id','$name');";
        'MySQLInsert $architecture' "localhost" "saat" "Saat01" "$query";
    }
}
```
File: fillS.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Scenarios in the xmi file (INPUT_FILE) are stored in de database written in DATABASE_FILE_NAME.

```perl
#!/c:/perl/bin/perl

# -- extract Scenarios from xmi-file and store them in text file using following format
ID,NAME
\..\\Parser\\extractScenarios.pl $ARGV[1] ..\\Parser\\Output\\S.txt
  ..\\Parser\\Output\\Parser.log

# -- open file containing database name
open(Arch_FILE,$ARGV[0]);
$architecture = '<ARCH_FILE>);

# -- open file containing scenarios
open(S_FILE,'..\\Parser\\Output\\S.txt');

# -- each line contains a scenario (record for table S)
@lines = <S_FILE>;
foreach $line (@lines) {
    # -- line is empty ?
    if($line)
    {
        # -- split line at , Two fields remain ID and NAME
        @fields = split ',',$line,2;
        $id = $fields[0];
        $name = $fields[1];
        chomp($id);
        chomp($name);

        # -- store into table S using MySQLInsert
        $query = 'INSERT INTO S (ID,NAME) VALUES(${id},${name});');
        'MySQLInsert "$architecture" "localhost" "saat" "Saat01" "$query"
    }
}
```
File: fillC.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Components in the xmi file (INPUT_FILE) are stored in the database written in DATABASE_NAME_FILE.

```perl
#!/c/perl/bin/perl

# -- extract Components from xmi-file and store in text file using following format ID,NAME
'../Parser\extractComponents.pl $ARGV[1] ../Parser\Output\C.txt

# -- open file containing database name
open(ARCH_FILE,$ARGV[0]);
$architecture = <ARCH_FILE>;

# -- open file containing components
open(C_FILE,"../Parser\Output\C.txt");

# -- each line is record for table C {component}
@lines = <C_FILE>;
foreach $line (@$lines)
{
    # -- empty line?
    if($line)
    {
        # -- split into fields at ,
        @fields = split /,,,$line,2;
        $id = $fields[0];
        $name = $fields[1];
        chomp($id);
        chomp($name);

        # -- store into table C using MySQLInsert
        $query = "INSERT INTO C (ID,NAME) VALUES('$id','$name');"
        'MySQLInsert "$architecture" 'localhost' "seal" "Seat01" "$query";
    }
}
```
File: fillM.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Services in the xmi file (INPUT_FILE) are stored in the database written in DATABASE_FILE_NAME.

#!/c/perl/bin/perl

# -- extract Services to text file using the following format ID,NAME
`..\Parser\extractServices.pl $ARGV[1] ..\Parser\Output\M.txt`

# -- open file containing database name
open(ARCH_FILE,<$ARGV[0]>

#architecture = <ARCH_FILE>

# -- open file containing Services
open(M_FILE,`..\Parser\Output\M.txt`

# -- each line is Service (record for table M)
@lines = <M_FILE>

foreach $line (@lines)
{
  # -- is line empty?
  if($line)
  {
    # -- split line into fields at ,
    @fields = split ',',$line,2;
    $id = $fields[0];
    $name = $fields[1];
    chomp($id);
    chomp($name);

    # -- store into table M using MySQLInsert
    $query = "INSERT INTO M (ID,NAME) VALUES('$id','$name');"
    "MySQLInsert "$architecture" "localhost" "sat" "Sat01" "$query";
  }
}
File: fillT.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- States in the xmi file (INPUT_FILE) are stored in the database written in DATABASE_FILE_NAME.

```perl
#!/c:/perl/bin/perl

# -- extract States from xmi-file to text file using following format ID,NAME
# ...\Parser\extractStates.pl $ARGV[1] ...\Parser\Output\T.txt
# ...\Parser\Output\Parser.log

# -- open file containing database name
open(ARCH_FILE,$ARGV[0]);
$architecture = <ARCH_FILE>

# -- open file containing states
open(T_FILE, "...\Parser\Output\T.txt");

# -- each line contains record for table T
@lines = <T_FILE>
foreach $line (@lines)
{

    # -- is line empty?
    if($line)
    {

        # -- split line into fields at ,
        @fields = split /,,,$line,2;
        $id = $fields[0];
        $name = $fields[1];
        chomp($id);
        chomp($name);

        # -- insert into T using MYSQLInsert
        $query = "INSERT INTO T (ID,NAME) VALUES('$id','$name');"
        'MYSQLInsert "$architecture" "localhost" "saat" "Saat01" "$query"';
    }
}
File: fillUS.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Use cases – scenario relations in the xmi file (INPUT_FILE) are stored in database written in DATABASE_FILE_NAME.

```perl
#!/usr/bin/perl

# -- extract UseCase Scenario relation from xmi-file to text file
# -- using following format ID_U, ID_S
  `..\Parser\extractUs.pl \$ARGV[1] ..\Parser\Output\Us.txt ..\Parser\Output\Parser.log`;

# -- open file containing database name
open(ARCH_FILE, \$ARGV[0]);
$architecture = <ARCH_FILE>;

# -- open file containing relations
open(Us_FILE, "..\Parser\Output\Us.txt");

# -- each line contains record for table US
@lines = <Us_FILE>;
foreach $line (@lines)
{
  # -- is line empty?
  if($line)
  {
    # -- split lines into fields at ,
    @fields = split ',', $line;
    $id_u = $fields[0];
    $id_s = $fields[1];
    chomp($id_u);
    chomp($id_s);

    # -- insert into Table US using MySQLInsert
    $query = "INSERT INTO US (ID_U, ID_S) VALUES('$id_u', '$id_s');";
    "MySQLInsert "$architecture" "localhost" "saat" "Saat01" "$query";
  }
}
File: fillCM.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Component - Service relations in the xmi file (INPUT_FILE) are stored in database written in DATABASE_FILE_NAME.

```
#!/usr/bin/perl

# -- extract Component Service relation from xmi-file into text file using following
# -- format ID_C, ID_M
'..\Parser\extractCM.pl $ARGV[1] ..\Parser\Output\Cm.txt ..\Parser\Output\Parser.log';

# -- fetch database name
open(ARCH_FILE, $ARGV[0]);
$architecture = <ARCH_FILE>;

# -- open file containing record information
open(Cm_FILE, "..\Parser\Output\Cm.txt");

# -- each line contains information for record CM
@lines = <Cm_FILE>;
foreach $line (@lines)
{
    # -- is line empty?
    if($line)
    {
        # -- split line into fields at ,
        @fields = split /,/, $line, 2;
        $id_c = $fields[0];
        $id_m = $fields[1];
        chomp($id_c);
        chomp($id_m);

        # -- insert into CM using MySQLInsert
        $query = "INSERT INTO CM (ID_C, ID_M) VALUES('$id_c', '$id_m');";
        MySQLInsert "$architecture" 'localhost' 'saat' 'Saat01' "$query";
    }
}
```
File: fillSM.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Service calls in the xmi file (INPUT_FILE) are stored in the database written in DATABASE_FILE_NAME.

```
#!/usr/bin/perl

# -- extract service calls from xmi-file into text file using following format
# -- ID, ID_S, ID_C1, ID_C2, ID_M2, ID_PRED
'..\Parser\extractSm.pl $ARGV[1] ..\Parser\Output\SM.txt ..\Parser\Output\Parser.log';

# -- fetch database name
open(ARCH_FILE, '<ARCH_FILE>');
$architecture = <ARCH_FILE>;

# -- open text files containing service calls
open(SM_FILE, '..\Parser\Output\SM.txt');

# -- each line is element (record) for table SM
@lines = <SM_FILE>;
foreach $line (@lines)
{
    # -- is line empty?
    if($line)
    {
        # -- split line into fields at ,
        @fields = split ',', $line;
        $id = $fields[0];
        $id_s = $fields[1];
        $id_c1 = $fields[2];
        $id_c2 = $fields[3];
        $id_m2 = $fields[4];
        $id_pred = $fields[5];
        chomp($id);
        chomp($id_s);
        chomp($id_c1);
        chomp($id_c2);
        chomp($id_m2);
        chomp($id_pred);

        # -- insert into SM using MySQLInsert
        $query = "INSERT INTO SM (ID, ID_S, ID_C1, ID_C2, ID_M2, ID_PRED)
                VALUES('$id', '$id_s', '$id_c1', '$id_c2', '$id_m2', '$id_pred');";
        'MySQLInsert $architecture "localhost" "saat" "Saat01" "$query";
    }
}
```
File: fillCT.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Component State relations in the xmi file (INPUT_FILE) are stored in the database written in DATABASE_NAME_FILE.

```perl
#!/c:/perl/bin/perl

# -- extract Component State relations from xmi-file into text file using following format
# -- ID.C.ID.T
# ..\Parser\ExtractCt.pl $ARGV[1] ..\Parser\Output\Ct.txt ..\Parser\Output\Parser.log'

# -- fetch database name
open(ARCH_FILE, $ARGV[0]);
$architecture = <ARCH_FILE>

# -- open text file containing Component State information
open(Ct_FILE, '..\Parser\Output\Ct.txt');

# -- each line is record for table CT
@lines = <Ct_FILE>
foreach $line (@lines)
{
    # -- is line empty?
    if($line)
    {
        # -- split line into fields at ,
        @fields = split /,/$line,2;
        $id_c = $fields[0];
        $id_t = $fields[1];
        chomp($id_c);
        chomp($id_t);

        # -- insert into table CT using MySQLInsert
        $query = "INSERT INTO CT (ID.C,ID.T) VALUES($id_c,'$id_t');
        'MySQLInsert "$architecture" 'localhost' 'saat' 'Saat01' "$query";
    }
}
```
File: fillIT.pl
Arguments: <DATABASE_NAME_FILE, INPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DATABASE_NAME_FILE and INPUT_FILE are valid filenames.
- INPUT_FILE is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- State transitions in the xmi file (INPUT_FILE) are stored in the database written in DATABASE_FILE_NAME.

```perl
#!/usr/bin/perl

# -- extract state transitions from xmi-file into text file using following format ID_T1,ID_T2
# ..\Parser\extractTt.pl $ARGV[1] ..\Parser\Output\Tt.txt ..\Parser\Output\Parser.log;

# -- fetch database name
open(ARCH_FILE,$ARGV[0]);
$architecture = <ARCH_FILE>;

# -- open text file containing Transitions
open(TT_FILE, "..\Parser\Output\Tt.txt");

# -- each line contains record for table TT
@lines = <TT_FILE>;
foreach $line (@lines)
{
    # -- is line empty?
    if($line)
    {
        # -- split line into fields at ,
        @fields = split /,,$line,2;
        $id_t1 = $fields[0];
        $id_t2 = $fields[1];
        chomp($id_t1);
        chomp($id_t2);

        # -- insert into table TT using MySQLInsert
        $query = "INSERT INTO TT (ID_T1,ID_T2) VALUES('$id_t1','$id_t2');";
        MySQLInsert "$architecture" "localhost" "saat" "Saat01" "$query";
    }
}
```
Software Architecture Analysis Tool

File: MySQLInsert.cpp
Arguments: <DB,HOST,USER,PASSWD,QUERY_STRING>
Language: Visual C++ 6.0
Author: Johan Muskens

Implementation for component: DatabaseFiller

Pre-condition:
- DB, HOST, USER, PASSWD give access to database.
- Table to which data will be added exists
- QUERY_STRING is SQL query that inserts data into table of DB
- LibMySQL.dll is in the dll-path

Post-condition:
- Data is added to table of DB.

// MySQLInsert.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include "MySQLInsert.h"

#include <iostream>
#include <iomanip>
#include <mysql++>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
static char THIS_FILE[] = __FILE__;
#endif

.isRequired

// The one and only application object

CWinApp theApp;
using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    int nRetCode = 0;
    // initialize MFC and print and error on failure
    if (!AFXWinInit::GetModuleHandle(NULL, NULL, ::GetCommandLine(), 0))
    {
        // TODO: change error code to suit your needs
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
    } else
    {
        try {
            // fetch program arguments DB, HOST, USER, PASSWD, QUERY
            string db = "";
            string host = "";
            string user = "";
            string passwd = "";
            string query_string = "";
            if(argc < 6)
            {
                return -1;
            }
            else
{  
db = argv[1];  
host = argv[2];  
user = argv[3];  
passwd = argv[4];  
query_string = argv[5];  
}

// make connection  
Connection  
con(db.c_str(),host.c_str(),user.c_str(),passwd.c_str());

// create query for insertion  
Query query = con.query();  
query << query_string;;

// execute query  
Result res = query.store();

}  
catch (BadQuery er)  
{
  cerr << "Error: " << er.error << endl;  
nRetCode = -1;
}

catch (BadConversion er)  
{
  cerr << "Error: Tried to convert \" << er.data << "\" to a \"" << er.type_name << \"\"." << endl;
  nRetCode = -1;
}

return nRetCode;
File: checkDatabase.pl
Arguments: <DATABASE_NAME_FILE, CHECK_FILE, OUTPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseChecker

Pre-condition:
- DATABASE_NAME_FILE, CHECK_FILE and INPUT_FILE are valid filenames.
- Checks in CHECK_FILE are valid SQL statements on database in DATABASE_NAME_FILE.

Post-condition:
- Results of the checks are written to OUTPUT_FILE.

```
#!/c:/perl/bin/perl

# -- open files
open(ARCH_FILE, $ARGV[0]);
open(OUTPUT, '>'.$ARGV[2]);
close(OUTPUT);

$architecture = <ARCH_FILE>;

# -- read checks (SQL statements and names)
$check_file = $ARGV[1];
open(CHECK_FILE, $check_file);
@checks = <CHECK_FILE>;

# -- execute each check
foreach $check (@checks) {
  # -- split name from sql-statement
  @name_query = split /\./, $check;
  $name = $name_query[0];
  $query = $name_query[1];
  chomp($query);

  # -- name or quere empty?
  if($query && $name) {
    # -- output name of check
    open(OUTPUT, '>'.$ARGV[2]);
    print OUTPUT "[$name]\n";
    close(OUTPUT);

    # -- execute check ....
    print 'executeCheck.pl '$architecture' "$query" "$ARGV[2]"
; $query = "";
    $name = "";
    open(OUTPUT, '>'.$ARGV[2]);
    print OUTPUT "\n\n";
    close(OUTPUT);
  }
}
```
File: executeCheck.pl
Arguments: <DATABASE_NAME, QUERY, OUTPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: DatabaseChecker

Pre-condition:
- DATABASE_NAME and QUERY are valid.

Post-condition:
- Results of the query are written to OUTPUT_FILE.

```perl
#!/c:/perl/bin/perl

# -- fetch arguments
$architecture = $ARGV[0];
$query = $ARGV[1];

# -- execute check
open(OUTPUT, ">>$ARGV[2]");
print OUTPUT "MySQLQuery "$architecture" "localhost" "saat" "Saat01" "$query"
; close(OUTPUT);
```
Pre-condition:
- HOST, USER, PASSWD give acces to DATABASE_NAME.
- QUERY is a valid SQL statement
- LibMySQL.dll is in the dll-path
Post-condition:
- Results of the query are written to std out.

```
#include "stdafx.h"
#include "MySQLQuery.h"

#include <iostream>
#include <iomanip>
#include <mysql++>

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
static char THIS_FILE[] = __FILE__;
#endif

// The one and only application object
CWinApp theApp;

using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    int nRetCode = 0;

    // initialize MFC and print and error on failure
    if (!AfxWinInit(NULL, NULL, //GetModuleHandle(NULL), 0))
    {
        // TODO: change error code to suit your needs
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
    }
    else
    {
        try {
            // fetch arguments DB, HOST, USER, PASSWD, QUERY_STRING
            string db = "";
            string host = "";
            string user = "";
            string passwd = "";
            string query_string = "";
            if(argc < 6)
            {
                return -1;
            }
            else
            {
```
db = argv[1];
host = argv[2];
user = argv[3];
passwd = argv[4];
query_string = argv[5];

// make connection to database
Connection
    con(db.c_str(), host.c_str(), user.c_str(), passwd.c_str());

// create query
Query query = con.query();
query << query_string;

// execute query
Result res = query.store();

// output result
Row row;
Result::iterator i;
for (i = res.begin(); i != res.end(); i++)
{
    // output row (record)
    row = *i;
    bool first = true;
    Row::iterator j;
    for(j = row.begin(); j != row.end(); j++)
    {
        // output column (field)
        if(!first)
        {
            cout << ",";
        }
        else
        {
            first = false;
        }
        cout << *j;
    }
    cout << endl;
}
catch (BadQuery er)
{
    cerr << "Error: " << er.error << endl;
    nRetCode = -1;
}
catch (BadConversion er)
{
    cerr << "Error: Tried to convert " << er.data << " to a " << er.type_name << "рус" << endl;
    nRetCode = -1;
}
return nRetCode;
File: analyse.pl
Arguments: <INPUT>
Language: Perl
Author: Johan Muskens

Implementation for component: Analyser

Pre-condition:
- INPUT is valid filename.
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools

Post-condition:
- Results of the analysis will be written to file result.txt written in the output directory of the Analyser.

```perl
#!/c:/perl/bin/perl

# -- create database
print "[creating database]\n";
print ".\n";
'DatabaseCreator\MySQLCreateDb localhost saat01 > DatabaseCreator\Output\Db.txt';

# -- create error file
open(ERROR,">Parser\Output\Parser.log");
$time = localtime(time);
print ERROR "----- $time ----\n";
close(ERROR);

# -- fill database
print "[filling database]\n";
chdir "DatabaseFiller";
print "filling Un\n";
'fillU.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Sn\n";
'fillS.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Cn\n";
'fillC.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Mn\n";
'fillM.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Tn\n";
'fillT.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Us\n";
'fillUs.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Sm\n";
'fillSm.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Cm\n";
'fillCm.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Ct\n";
'fillCt.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';

print "filling Tt\n";
'fillTt.pl \nDatabaseCreator\Output\Db.txt $ARGV[0]';
```
chdir "..");

# -- check database
print "[checking database]\n";
print ".\n";
chdir "DatabaseChecker";
'checkDatabase.pl ..\DatabaseCreator\Output\Db.txt check.txt Output\checkresult.txt';
chdir "..");

# -- analyse database
print "[analysing database]\n";
print ".\n";
chdir "Analyser";
'analyseArchitecture.pl ..\DatabaseCreator\Output\Db.txt analysis.txt Output\result.txt';
chdir "..");
File: analyseArchitecture.pl
Arguments: <DATABASE_NAME_FILE, ALGORITHM_FILE, OUTPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: Analyser

Pre-condition:
- DATABASE_NAME_FILE, ALGORITHM_FILE and OUTPUT_FILE are valid filenames.

Post-condition:
- Results of the checks in the ALGORITHM_FILE applied on the database described in DATABASE_NAME_FILE are written to OUTPUT_FILE.

```perl
#!/c:/perl/bin/perl

# -- open files
open(ARCH_FILE, $ARGV[0]);
open(OUTPUT, ">$ARGV[2]");
close(OUTPUT);

$architecture = <ARCH_FILE>;

# -- read algorithms / metrics (SQL statements)
$algorithm_file = $ARGV[1];
open(ALG_FILE,$algorithm_file);
@algorithms = <ALG_FILE>;

# -- execute algorithms
foreach $algorithm (@algorithms) {
    # -- split name from sql statement
    @name_query = split /;/,$algorithm,2;
    $name = $name_query[0];
    $query = $name_query[1];
    chomp($query);

    # -- query or name empty?
    if($query && $name) {
        # -- output name
        open(OUTPUT, ">$ARGV[2]");
        print OUTPUT "$name\n";
        close(OUTPUT);

        # -- execute query and output results
        print \'executeAnalysis.pl \'$architecture\' \'$query\' \'$ARGV[2]\';
        $query = "";
        $name = "";
        open(OUTPUT, ">$ARGV[2]");
        print OUTPUT \"\n\n";
        close(OUTPUT);
    }
}
```
File: executeAnalysis.pl
Arguments: <DATABASE_NAME, ALGORITHM, OUTPUT_FILE>
Language: Perl
Author: Johan Muskens

Implementation for component: Analyser

Pre-condition:
- DATABASE_NAME and ALGORITHM are valid.

Post-condition:
- Results of the check applied on the database DATABASE_NAME are written to OUTPUT_FILE.

```perl
#!/usr/bin/perl

$architecture = $ARGV[0];
$query = $ARGV[1];

open(OUTPUT, " >> $ARGV[2]");
print OUTPUT \MySQLQuery " $architecture " "localhost" "saat" "Saat01" "$query";
close(OUTPUT);
```
File: filter.pl
Arguments: <INPUT, OUTPUT>
Language: Perl
Author: Johan Muskens

Implementation for component: StatisticFilter

Pre-condition:
- INPUT and OUTPUT are valid filenames.

Post-condition:
- Content of INPUT is filtered and the outlying values are written to output.

```perl
#!/c:/perl/bin/perl

# -- create statistics
print "[creating statistics]\n";
print "\n";
'StatisticsCalculator\calcStats.pl $ARGV[0] StatisticsCalculator\Output\stats.txt';

# -- filter results
print "[filtering results]\n";
print "\n";
'\ResultFilter\filterOnStats.pl $ARGV[0] StatisticsCalculator\Output\stats.txt $ARGV[1]';
```
File: calcStats.pl  
Arguments: <INPUT,OUTPUT>  
Language: Perl  
Author: Johan Muskens  

Implementation for component: StatisticCalculator  

Pre-condition:  
- INPUT and OUTPUT are valid filenames.  

Post-condition:  
- For the results in INPUT the average, variance, deviation and number of elements are calculated and written in OUTPUT.

```perl  
#!c:/perl/bin/perl  
# -- open input file (file with results of analysis)  
open(INFILE, ">ARGV[0]");  
open(OUTPUT, ">ARGV[1]");  
@lines = <INFILE>;

$first = 0;
foreach $line (@lines)  
{
    chomp($line);
    # -- result of new metric?  
    if($line =~ /\s*\{\s*\}/gi)
    {
        if($first != 0 && $count != 0)
        {
            # -- calculate average, variance and deviation  
            $average = $sum / $count;
            $tmp_value = 0;
            foreach $score (@scores)
            {
                $tmp_value = $tmp_value + (($score - $average)*
                ($score - $average));
            }
            $variance = $tmp_value / $count;
            $deviation = sqrt $variance;
            # -- output stats  
            print OUTPUT "$name\n";
            print OUTPUT "Count:$count\n";
            print OUTPUT "Average:$average\n";
            print OUTPUT "Variance:$variance\n";
            print OUTPUT "Standard deviation:$deviation\n\n";
        }
        $variance = $tmp_value / $count;
        $deviation = sqrt $variance;
        # -- output stats  
        print OUTPUT "$name\n";
        print OUTPUT "Count:$count\n";
        print OUTPUT "Average:$average\n";
        print OUTPUT "Variance:$variance\n";
        print OUTPUT "Standard deviation:$deviation\n\n";
    }
    # -- initialize values for new results of next metric  
    $first = 1;
    $name = $1;
    $count = 0;
    $sum = 0;
    @scores = ();
    else
    {
        # -- empty line?  
        if($line)
        {
            # -- add score  
            @tmp = split /\./,$line;
        }
    }
```
Software Architecture Analysis Tool

```bash
$scores[$count] = $tmp[1];
$sum = $sum + $tmp[1];
$count++;
}

if($first != 0 && $count != 0)
{
    $average = $sum / $count;
    $tmp_value = 0;
    foreach $score (@scores)
    {
        $tmp_value = $tmp_value + (($score - $average)*($score - $average));
    }
    $variance = $tmp_value / $count;
    $deviation = sqrt $variance;

    print OUTPUT "[Name]\n";
    print OUTPUT "Count:$count\n";
    print OUTPUT "Average:$average\n";
    print OUTPUT "Variance:$variance\n";
    print OUTPUT "Standard deviation:$deviation\n";
}
```
File: filterOnStats.pl
Arguments: <INPUT, STATS, OUTPUT>
Language: Perl
Author: Johan Muskens

Implementation for component: StatisticFilter

Pre-condition:
- INPUT, STATS and OUTPUT are valid filenames.

Post-condition:
- Content of INPUT is filtered based on STATS, the outlying values are written to OUTPUT.

```perl
#!/c:/perl/bin/perl

# -- open files
open(INPUT, $ARGV[0]);
open(STATS, $ARGV[1]);
open(OUTPUT, ">$ARGV[2]");

# -- read input
@lines = <INPUT>;

# -- rend stats
@stat_lines = <STATS>;

$average = 0;
$deviation = 0;
foreach $line (@lines)
{
  # -- new algorithm?
  if($line =~ /[\{\}\[\]\{\}\]|\d+\d+\d+\d+)/gi)
  {
    # -- fetch average and deviation
    $found = 0;
    $name = $1;
    foreach $stat_line (@stat_lines)
    {
      $tmp = $stat_line;
      if($tmp =~ /\{\}\[\]\{\}\]|\d+\d+\d+\d+)/gi)
      {
        if($1 =~ /$name/gi)
        {
          $found = 1;
        }
        else
        {
          $found = 0;
        }
      }
      else
      {
        if($found == 1)
        {
          if($stat_line =~ /Average/gi)
          {
            @fields = split /,\/$stat_line, 2;
            $average = $fields[1];
          }
          if($stat_line =~ /Standard deviation/gi)
          {
            # code for standard deviation
          }
        }
      }
    }
  }
  else
  {
    # code for new algorithm
  }
```
{ $fields = split /\,/,$stat_line,2; $deviation = $fields[1];

} } print OUTPUT $line;
} else {
  # -- empty line?
  $tmp2 = $line;
  chomp($tmp2);
  if($tmp2) {
    # -- check whether score of current line is outlying value
    $fields = split /\,/,$line,2;
    $bottom = $average - (2 * $deviation);
    $top = $average + (2 * $deviation);
    if($fields[1] < $bottom || $fields[1] > $top) {
      # -- in case of outlying value output the line
      print OUTPUT $line;
    }
  } else {
    # -- empty lines are also copied
    print OUTPUT "\n";
  }
} }


File: a2html.pl
Arguments: <INPUT,OUTPUTDIR>
Language: Perl
Author: Johan Muskens

Implementation for component: none (just added to make results more fancy)

Pre-condition:
- INPUT is valid result file.
- OUTPUTDIR is valid directory for output of the html-files

Post-condition:
- Results of analysis in plain ascii are converted into html, for each algorithm a separate file is create and an entry in a index file is added.

```perl
#!/c:/perl/bin/perl

# -- open input and index file
open(INFILE,$ARGV[0]);
open(INDEX,">$ARGV[1]algorithms.html");

# -- read input file
@lines = <INFILE>;

# -- output html header
print INDEX "<html><n";
print INDEX "<head><n";
print INDEX "<title>Analysis report</title><n";
print INDEX "</head><n";
print INDEX "<body><n";
print INDEX "<ul><n";

# -- process all lines of input
$first = 0;
$elements = 0;
$algs = 0;
foreach $line (@lines)
{
    chomp($line);
    # -- new algorithm?
    if($line =~ /\{\{\{\}\}\}\}/gi)
    {
        $algs++;
        # -- print footer if necessary
        if($first != 0)
        {
            if($elements == 0)
            {
                print RESULT "<tr><td>no elements</td></tr><n"
            }
            print RESULT "</table><n"
            print RESULT "</body><n"
            print RESULT "</html><n"
        }
        $first = 1;
        $elements = 0;

    # -- create entry in index for this algorithm
    }print INDEX "<li><a href="$algs.html" target="results">$1</a></li><n"
    # -- create new file for results of this algorithm
    open(RESULT,">$ARGV[1]$algs.html");
```
# -- output html header
print RESULT "<html>
";
print RESULT "<head>
";
print RESULT "<title>Analysis report</title>
";
print RESULT "</head>
";
print RESULT "<body>
";
print RESULT "<h2>
";
print RESULT "<table border='1'>
";

}
else
{

    # -- empty line?
    if($line)
    {
        # -- add element to html file (add to table)
        $elements++;
        $tmp = join '</td><td>', split '\n', $line;
        print RESULT "<tr><td>$tmp</td><tr>
";
    }
}

if($first != 0)
{
    if($elements == 0)
    {
        print RESULT "<tr><td>no elements</td><tr>
";
    }
    print RESULT "</table>
";
    print RESULT "</body>
";
    print RESULT "</html>
";
}

print INDEX "</ul>
";
print INDEX "</body>
";
print INDEX "</html>
";}
File: saat.bat
Arguments: <INPUT>
Language: Batch file
Author: Johan Muskens

Implementation for component: SAAT

Pre-condition:
- INPUT file is filename of model exported to UML with Rational Rose UniSysRoseXMLTools.

Post-condition:
- Model is analysed and the results can be found in the output directory of SAAT.

echo off
cls

analyse.pl %1

filter.pl Analyser\Output\result.txt ResultFilter\Output\filtered.txt

echo [creating html files]
echo .
a2html.pl Analyser\Output\result.txt Output\UnFiltered\Results\a2html.pl ResultFilter\Output\filtered.txt Output\Filtered\Results\