CHALMERS

Java exception matching in real time using fuzzy logic

Master of Science Thesis

Karl Tillström

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2010

Java exception matching in real time using fuzzy logic

KARL TILLSTRÖM

Examiner: BROR BJERNER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden May 2010

Abstract

This thesis deals with the problem of matching Java exceptions that originates from the same problem, but looks a bit dissimilar. The matching must be done within reasonable time to be able to be carried out automatically. Different ways of utilizing fuzzy string matching logic is examined in general and how to apply it to Java exceptions specifically.

The exceptions are divided into separate logical parts and the analysis deals with how to weight the importance of the parts and application of fuzzy matching between them. Java exceptions can be linked in “caused by” chains depending on where they occur and what effects they have. In short - an exception can cause other exceptions to occur, and thus the first exception is a “caused by exception” to the second one.

In this thesis the resulting algorithm groups exceptions together with their “caused by”:s and compares the top and bottom exceptions of the chain with other top and bottom exceptions to find matches. To determine degree of similarity a number of fuzzy string matching algorithms where examined including “Levenshtein”, “Damerau-Levenshtein”, “Needleman-Wunch”, “Jaro-Winkler”, “Bitap”, “Boyer-Moore” and in some degree “Ukkonen”. In the end, a modified version of the original Levenshtein algorithm - utilizing lazy evaluation and thresholding - was determined the best suited comparison algorithm for this problem.

Abstract

Den här examenstesen berör problemet om att matcha avbrott i Java (eng: exceptions) som härrör från samma problem, men av olika anledningar ser lite olika ut. Matchningen måste göras inom rimlig tid för att kunna användas automatiskt av andra applikationer i realtid. För att lösa detta undersöks olika sätt att applicera s.k. fuzzy string matching-algoritmer på Java-avbrott.

Java-avbrott är uppdelade i olika logiska delar och analysen handlar om hur man kan vikta hur viktiga de olika delarna är och hur fuzzy string matching kan användas för att få fram likheter. Java-avbrott kan även vara ihoplänkade i orsaks-kedjor (eng: caused by) beroende på var de uppstår och vilka effekter de har. Kortfattat kan ett avbrott orsaka att ett annat uppstår och då blir det första avbrottet ett orsaksavbrott till det andra.

I den här tesen grupperas avbrott ihop med deras orsakande avbrott och det översta i kedjan sätts ihop med det understa och jämförs med andra topp- och bottenavbrott. För att avgöra graden av likhet mellan de olika delarna har ett antal “fuzzy string matching”-algoritmer jämförts: “Levenshtein”, “Damerau-Levenshtein”, “Needleman-Wunch”, “Jaro-Winkler”, “Bitap”, “Boyer-Moore” och i viss mån “Ukkonen”. Slutligen har en modifierad variant av Levenshteinalgoritmen visats vara bäst lämpad för det här problemet - en Levenshtein med modifikationen att den använder “Lazy evaluation” samt ett tröskelvärde för hur olika strängar får vara som mest innan de kastas bort.

Contents

1 Preface

This report is a part of a master thesis project in Computer science and Engineering at the department of Computer Science, Chalmers University of Technology, Gothenburg, Sweden. The project was performed over a period of 8 months at the company Amadeus located at Sophia Antipolis, France and marks the end of the authors studies to a Master of Science in Computer Science and Engineering.

First off all I would like to thank Amadeus for the great opportunity and all efforts to support my work – in particular Raphael Kubler – my tutor, the Production Support team (PSU) – his team and also Hugo Questroy and Laurent Cognard from the NRE team for their excessive support of the project. I would also like to give a special thanks to “the Swedish mafia” at Amadeus for great company and quick integration into the French life - in particular Tobias “Tumm” Engvall for convincing me, that a master thesis in France was the way to go. At Chalmers I would like to thank Bror Bjerner for supervising the project in a very non problematic way, in spite of French bureaucracy’s efforts to make things complicated.

Last but not least, I would like to thank Meea for support on everything else in life during this Master thesis!

2 Introduction

This thesis is about matching Java exceptions with each other to determine degree of similarity between them and to do so with acceptable performance. An exception origins from a certain point in the source code. However depending on different circumstances such, as version of code, platform, and machine type - the very same exception generated from the exact same source code, doesn't look exactly the same each time. Therefore, no simple check of exact equality can be used for exception matching. Exceptions originating from the same point of source should of course be considered to be equal even though they might differ slightly in appearance. The matching is thus to be done by first defining which features that are relevant in an exception and then by constructing logical rules and algorithms to calculate the resemblance given those features. A proof of concept application should be constructed, that executes the task with reasonable performance to be able to be used in daily work in a big production environment.

Amadeus has several big web based systems running on WebLogic platforms, which produce a huge amount of exception logs every day. Due to the complexity of programming and system development, exceptions are an inevitable effect and most often indicate some error either in code or in design. Encountered exceptions must first be recognized as either a previously unknown exception or an exception in the process of being resolved, before any action should be taken to correct the error. Given a big system in production, this is infeasible to carry out manually, due to the sheer amount of exception logs produced every day. It is also problematic to carry out automatically, since fuzzy matching algorithms tend to be heavy performance wise, which is one major issue to solve within this thesis.

To keep track of exceptions (and other problems), Amadeus uses "Problem tracking records" or in short: PTRs. The goal of the program developed to implement the Java Exception matching, is thus to be able to match given exceptions to existing PTRs regarding the same, previously encountered exception.

Amadeus has also developed internal frameworks and other software standards that new developments should be implemented in conjunction with. There are particularly two such internal systems the developed application will be dealing with: SPIN[1] and SWAT[2]. SPIN is a Java framework with additional policies and SWAT is an automated testing tool and data storage, which deals a lot with exceptions. Hence, the exception model developed here will also be usable by SWAT.

2.1 Purpose

The problem to solve is how a method of matching Java Exceptions can be developed that utilizes the specific features of the Java Exception structure and syntax and is possible to implement and be used in a production environment with high data flows, *in real time*. Two exceptions that are considered to be equal are not necessarily identical and must thus be matched based on particular rules regarding their properties/features and some variation of fuzzy logic. What

[1]SPIN is a framework and set of policies developed internally within Amadeus
[2]SWAT is a data collection and analysis project within Amadeus

these rules are is a part of the problem and must be developed along with the logic that should implement them for a complete working solution. This implies a trade off between accuracy and performance and a big part of the problem is to find a method that is usable with good enough accuracy and performance in practice and not just in theory.

To solve this problem, different rules, logic and algorithms will be analyzed and tested and as the final result a complete application for the task will be constructed based on the results of the analysis and testing. The application should be developed using a modular approach and by multiple possibilities of user/application interfaces, to be able to be used by different platforms and teams within Amadeus. Different matching rules of the exception features will be tried out and tested, to find one that gives a result, that is useful for daily work with exception logs. Different algorithms for fuzzy string matching will be evaluated to find one that performs well enough to manage the performance requirements – e.g exception matching should appear to be done in "no time" for the user, regardless of the amount of exceptions kept track of. A data model for storing the results will be evaluated and implemented with respect to storage size, performance requirements, and data integrity.

2.2 Delimitation

In its context, the definition of the problem yields a couple of constraints:

- Only WebLogic generated exceptions are guaranteed to be compliant with the application.
- Since the matching part is the only relevant function of the application, only the features of an exception needed for matching are stored, i.e. that no complete exceptions are stored and thus the application may need to be complemented with some storage of complete exceptions for problem investigation.
- The application is an interface for single exception matching – thus no form of parser of log files or likewise is implemented.

I know of no other existing exception matching applications and thus the prior work done in the field that is applicable to this project, is mainly general fuzzy string matching.

2.3 Original project definition

Exception application tracker

The internship will consist of two main studies:

1. Functionality:

The trainee will examine a way to store exception in the database with an associated problem record.

This study will involve the following aspects:

- Exception stake trace management

- A algorithm to compare an exception to other exception and provide a percentage matching number
- Impacted Production systems
- Referenced Problem records
- Automatic Problem record creation for new error/exception

2. GUI:

The trainee will study the development of a Left/Right UI window to allow people to compare an exception in the log file with one stored in the database. The trainee will also study the proposal of a web service to allow other application to invoke the service in batch mode to identify already existing exception

3 Analysis and methodology

The analysis and methodology is divided in eight sub-chapters, each describing a different building block of the master thesis. Since many of the parts depend on the progress made in other parts, the analysis and methodology section is not chronological, except for each sub-chapter itself.

3.1 Overall project plan

Before the project was started, a plan needed to be created that defined each step necessary to reach the goal of the completed Master thesis.

3.1.1 Methodology

Since the project is quite vaguely defined; I expected that the project definition and constraints would evolve during the course of the project and thus no standard development process was chosen. Mainly to be able to spend time on progressing forward instead of trying to adapt the reality to the model when more information was discovered and defined. However, a project plan is needed to be able to produce a solution that solves the actual problem and does so in time and with a good result.

To construct a good project plan, I analyzed the project description and constructed design documents and presentations based on several processes and methods. The design documents had suggestions on strategy, definitions, and constraints of the project and the process to create it. Feedback on the document contents was given by my tutor, my team and other parties involved. The design documents where to be quite blunt and general at first, with each part more thoroughly specified as the development reached it.

3.1.2 Analysis and result

Exception definition and parser A good feel for the knowledge in the area of fuzzy matching was needed to come up with a good plan of how the project should be implemented. If there were particularly good methods and algorithms that needed the data to be handled in a certain way, it is easier to build the application in a way that supports it from the beginning than trying to adapt it after a while. A bunch of algorithms were studied (found in the algorithms section) and also how to break down an exception into features suitable for string matching (found in the exception definition section). In addition to string matching, an idea of automatic clustering of exceptions through a self organizing artificial neural network such as a Kohonen map[3] was thought of but not investigated further. Amadeus keeps problem tracking records (PTRs) to which the exceptions were to be associated with. It was quickly found that there were really no feasible way of searching the PTR data storage for existing exception PTRs, so in order to associate the exceptions the application had to keep a data storage of its own. The application data storage started out empty and was successively filled with exceptions as they were discovered.

This lead up to a first suggestion of the order to analyze the different parts of the master thesis, a program structure, and an implementation phase plan.

[3]http://en.wikipedia.org/w/index.php?title=Self-organizing_map&oldid=206987930

Algorithm 1 General project plan

```
Part 0 – Preparation
    • Define what constitutes an Exception (feature extraction)
    • Study Spin & Swat for possible integration
    • Finalize a working copy of the design document
    • Study unit tests
Part 1 – Test and development framework
    • Java Interfaces for all modules
    • Simple log file parser
    • Local Database
    • One classifier / PTR Matcher
    • Simple UI (Java Application)
Part 2 – Algorithm trials
    • Internal benchmarker
    • Research, implement and try out different classifiers / matchers
    • Simple neural net trial for the severity classifier
Part 3 – Beta version
    • Migrate to final database solution
    • Real GUI (Web based?)
    • Web service
    • Optimize log parser
Part 4 – If there is time
    • PTR generator
    • Reports
    • Class ownership linkage
```

An additional idea was to try to automatically classify how severe the exceptions were by using some sort of self learning neural network. The classifier would learn to recognize the severity of exceptions better and better by getting feedback from the user and it would be a help to the team to prioritize the resolution of the exceptions. All needed information was to be kept in the data store for each exception, so a table of occurrences and a table of stack traces would be created.

A presentation with the main purpose of gathering feedback on what was needed from the application was held for representatives from all teams interested in the project. Two conclusions were drawn from the presentation. The first one that two existing Amadeus internal projects/frameworks; SPIN and SWAT; should be investigated with the intention of integrating this project with them. The other that it was inconclusive so far what interfaces there should be to the system, in addition to a web service and some sort of left/right GUI.

The SPIN project is an effort to collect the internal tools into one single project for easy sharing of code and functions and also to ensure the quality of the code through policies and development routines. The idea of this project is to make use of the library and also ensure the maintainability and further development through the SPIN routines.

For this project SWAT can be seen as an internal system mainly for log file and exception management and storage. The idea is to gather all exception related data in one single place and thus it would be preferable if this project can be integrated with SWAT.

To progress from this point, a design document was created to be used as a project definition for all parties to agree upon. The design document were to define:

- Description of the project Goals & Objective of the project
- In which context the project were to be used
- Major constraints to take into consideration
- Development methodology
- Time plan, split up in phases
- Requirements of the application
- Use cases
- User interfaces
- Program structure / System architecture
- Database model

In parallel to writing the design document; SPIN & SWAT integration was studied and a draft of a suitable database model was created. With the parts of the design document needed to progress through a new presentation/meeting done, a new presentation was held to decide upon whether to go with SPIN & SWAT, settle on the database model, decide which version of Java to use (since not all projects were up to date with the latest Java version), and also what types of user interfaces to develop. In short, the meeting decided to go with SPIN & SWAT, Java version 1.5, to use the proposed database model, and for user interface only use a web based one developed through another internal library called “Aria”. Aria is an AJAX based web GUI library developed at Amadeus.

With the new directives to use SPIN, a predefined set of project planning documents were to be used – and thus the original design document was never completed, but split up into the appropriate SPIN project documents instead. So now the project was managed by:

- Project description
  An overall description of the project

- Technical description
  The details on how the project should be implemented, including UML diagrams, database diagrams and related things.

- Road map
  A time plan for the implementation of the project.

While working on the SPIN documents; three main areas were investigated; Finalizing a SWAT compatible data model, defining how to work with stack traces of exceptions, and what algorithms that should be implemented for fuzzy string matching. For more information on the development, see the the appropriate sub chapters.

At this point, the internship subject felt to have grown a bit out of hand, so after a meeting with Raphael Kubler and Eric Durand it was decided to limit the project more to the original internship subject. In reality this meant to limit the data model to only include one copy of each exception and put the responsibility of exact recreation of exception occurrences on SWAT. This project should be using a database of its own, but the database should be easily linkable to SWAT. For each exception occurrence in SWAT, a PTR connection should be able to get by querying this project's database. After a meeting with Laurent Cognard, the team leader of the NRE team which are responsible for both the SPIN and the SWAT project, a new database model was decided upon and the responsibility of linking SWAT to this project was put on the NRE team[4]. The development of the database model can be followed in the database sub chapter.

At this point, the project was quite clearly defined and a phase consisting of mainly implementation started. The database model was implemented and appropriate stored procedures created, all according to Amadeus database policies. Definition of exception features and matching rules were developed through analyzing data, suggesting strategies to the team and getting feedback. The part that needed most adaptation, was the parser that should translate an exception, entered as a text blob, into an “exception object” used internally. The problem is that exceptions might look a bit different – they are intended for human interpretation and thus the syntax varies a lot. In the end regular expressions were created to parse each feature of the exception and cycles of testing the parser on more and more syntaxes was performed. After each cycle, the regular expression were updated according to the results and feedback. To be able to easily accept new syntaxes further on, the regular expressions were

[4]The NRE team is responsble for “non regression” testing

decided to be updateable through the application interface,. Algorithms and matching rules were implemented and benchmarked to find the optimal ones to use.

Due to the fact that SPIN was problematic to get up and running on the local machine, a new sub goal was added. A non web based prototype would be created to be able to test the ideas and get feedback from the team without having to wait for the SPIN installation to get up and running. Thus, a prototype was created using a Java Swing interface[5], which were tested by the team and a lot of feedback were given.

The final step was to get SPIN working, convert the Swing application into a web based one using SPIN libraries, and then to get it up and working. The details of the web based UI are found in the GUI sub chapter.

In the end, the Web service goal was cut from the project, since there was no time to implement it.

3.2 Exception definition

The cornerstone of the project is the Java exceptions encountered on a WebLogic platform and therefore all analysis depend on the definition of them. In short, an exception is "thrown" by a Java program when an error that is not handled occurs and it contains information of what type of error it is and where it occurred. WebLogic stores these exceptions in log files and they can look slightly different with a lot of different syntaxes and content.

```
####<May 13, 2008 5:27:04 AM GMT> <Error> <JRes> <mucwwp106> <aetmeurope2node7a>
<ExecuteThread: '18' for queue: 'JRES'> <kernel identity> <> <000000> <com.amadeus.ocg.standard.action.rulesdriven
.util.UtilitiesRulesDriven> <Error in tripPlanInModification;jsessionid=LpmpGKqnVTLfbQKnPnDwQVlSJx1JCwbxDB7GltfY1TZfGqw
nGnzd!-1436525840!1339964487!1210656318450> ava.lang.NullPointerException
      at org.apache.struts.tiles.definition.ComponentDefinitionsFactoryWrapper.getDefinition
(ComponentDefinitionsFactoryWrapper.java:84)
      at org.apache.struts.tiles.TilesRequestProcessor.processTilesDefinition(TilesRequestProcessor.java:152)
      at org.apache.struts.tiles.TilesRequestProcessor.processForwardConfig(TilesRequestProcessor.java:302)
      at org.apache.struts.action.RequestProcessor.process(RequestProcessor.java:229)
      at org.apache.struts.action.ActionServlet.process(ActionServlet.java:1194)
      at org.apache.struts.action.ActionServlet.doPost(ActionServlet.java:432)
      at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
      at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
      at weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction.run(ServletStubImpl.java:1072)
      at weblogic.servlet.internal.ServletStubImpl.invokeServlet(ServletStubImpl.java:465)
      at weblogic.servlet.internal.ServletStubImpl.invokeServlet(ServletStubImpl.java:348)
      at weblogic.servlet.internal.WebAppServletContext$ServletInvocationAction.run(WebAppServletContext.java:6981)
      at weblogic.security.acl.internal.AuthenticatedSubject.doAs(AuthenticatedSubject.java:321)
      at weblogic.security.service.SecurityManager.runAs(SecurityManager.java:121)
      at weblogic.servlet.internal.WebAppServletContext.invokeServlet(WebAppServletContext.java:3892)
      at weblogic.servlet.internal.ServletRequestImpl.execute(ServletRequestImpl.java:2766)
      at weblogic.kernel.ExecuteThread.execute(ExecuteThread.java:224)
      at weblogic.kernel.ExecuteThread.run(ExecuteThread.java:183)
```

Figure 1: Example of an exception

3.2.1 Methodology

The main problem in exception definition, is to find a definition of structure that covers all exception syntaxes and variations to be able to store, compare, and work with the exceptions. In order not to store too much data, it is also important to take into consideration, how much information content in an exception that is relevant enough to store. Storing too much would have an impact on both search performance, and storage size.

[5]Regular windows application

The method used was to analyze a number of exception logs, trials to find previous work done on the subject on the Internet, and writing prototypes and getting feedback from the production support team. The production support team, PSU, are working a lot with exceptions and is one of the intended recipients of the tool.

Finally, a prototype of a parser was built and it was extended incrementally to include all found exception syntaxes. Supplementary unit tests were written to make sure that the newly added syntax did not break the old definitions and by that, the final parser was evolved.

3.2.2 Analysis

A search on the Internet did not result in any findings of particular work done on the subject of exception classification, storing, and/or matching. Therefore, an ad hoc approach was used to come up with a usable exception definition.

The first approach was to mark a number of features in an exception and based on that, in addition to overall exception structure and syntax, trying to categorize different exceptions into a number of categories. If a practical number of possible categories were discovered, a data model representing the different categories was to be created.

The analyze yielded the exception feature definition as follows:

- Date and Time
  The date and time the exception was thrown
- Machine name
  The name of the server on which the executing code was run
- Class
  The class that the exception occurred in
- Class message
  A message that is bundled with the class
- Exception Type
  The type of the exception e.g. NullReferenceException
- Exception Message
  The message generated within the exception
- Caused by clauses
  If the exception is generated by other exceptions, the generating exceptions are stored as caused by clauses. Also known as "inner exceptions"
- jsessionID
  The session ID the user had at the web server when the exception occurred
- Relevant part of the stack trace
  Stack trace that refers to the application code and not to the platform (e.g. WebLogic)

A number of exceptions with the suggested features extracted, where sent to the Production Support Team for feedback. The conclusions resulted in that there were no particular categories that the exceptions could be divided into, but rather a model with the same features for all were proposed. The features could be either present or not present in an individual exception. Also the number of features where reduced to the ones that were concluded as the most relevant of the problem at hand. The reducing was motivated by, that the application should only match exceptions and not be part of the investigation or the solution of them.

Features decided to be relevant in an exception:

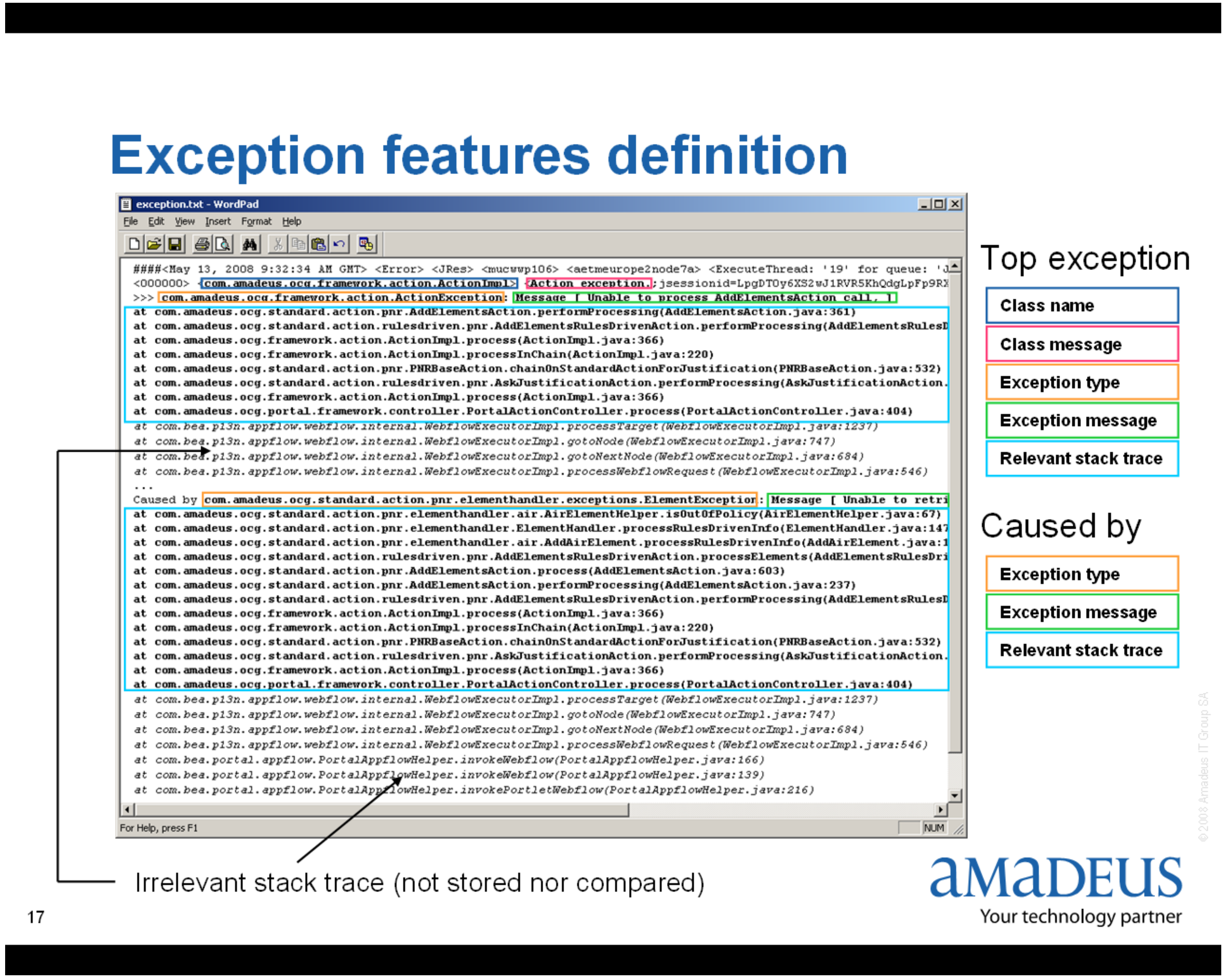
- Exception type
- Exception message
- Class name
- Class message
- Is a caused by
- Relevant Stack trace

Suggested features later concluded to be irrelevant:

- Date Time
- Machine name
- jSessionID

Whether or not each feature is a *mandatory* part of an exception had to be figured out. A first approach was to set Exception Type and Class Name as always present. This theory was later on discarded, since counter examples were discovered and in the end no feature was actually always present in all exceptions.

In addition to the features, the over all structure of an exception is defined as a "Top exception" with a stack trace followed by 0...n "Caused by exceptions". The caused by exceptions are strictly speaking different exceptions, but they are all linked together in an "exception chain". The "chain structure" may be important later on in the comparison algorithms.



Exception features definition

```
####<May 13, 2008 9:32:34 AM GMT> <Error> <JRes> <mucwwp106> <aetmeurope2node7a> <ExecuteThread: '19' for queue: 'J
<000000> <com.amadeus.ocg.framework.action.ActionImpl> <Action exception.>;jsessionid=LpgDTOy6XS2wJ1RVR5KhQdgLpFp9RX
>>> com.amadeus.ocg.framework.action.ActionException: Message [ Unable to process AddElementsAction call. ]
at com.amadeus.ocg.standard.action.pnr.AddElementsAction.performProcessing(AddElementsAction.java:361)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesDrivenAction.performProcessing(AddElementsRulesD
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.framework.action.ActionImpl.processInChain(ActionImpl.java:220)
at com.amadeus.ocg.standard.action.pnr.PNRBaseAction.chainOnStandardActionForJustification(PNRBaseAction.java:532)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AskJustificationAction.performProcessing(AskJustificationAction.
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.portal.framework.controller.PortalActionController.process(PortalActionController.java:404)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processTarget(WebflowExecutorImpl.java:1237)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNode(WebflowExecutorImpl.java:747)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNextNode(WebflowExecutorImpl.java:684)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processWebflowRequest(WebflowExecutorImpl.java:546)
...
Caused by com.amadeus.ocg.standard.action.pnr.elementhandler.exceptions.ElementException: Message [ Unable to retri
at com.amadeus.ocg.standard.action.pnr.elementhandler.air.AirElementHelper.isOutOfPolicy(AirElementHelper.java:67)
at com.amadeus.ocg.standard.action.pnr.elementhandler.ElementHandler.processRulesDrivenInfo(ElementHandler.java:147
at com.amadeus.ocg.standard.action.pnr.elementhandler.air.AddAirElement.processRulesDrivenInfo(AddAirElement.java:1
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesDrivenAction.processElements(AddElementsRulesDri
at com.amadeus.ocg.standard.action.pnr.AddElementsAction.process(AddElementsAction.java:603)
at com.amadeus.ocg.standard.action.pnr.AddElementsAction.performProcessing(AddElementsAction.java:237)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesDrivenAction.performProcessing(AddElementsRulesD
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.framework.action.ActionImpl.processInChain(ActionImpl.java:220)
at com.amadeus.ocg.standard.action.pnr.PNRBaseAction.chainOnStandardActionForJustification(PNRBaseAction.java:532)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AskJustificationAction.performProcessing(AskJustificationAction.
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.portal.framework.controller.PortalActionController.process(PortalActionController.java:404)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processTarget(WebflowExecutorImpl.java:1237)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNode(WebflowExecutorImpl.java:747)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNextNode(WebflowExecutorImpl.java:684)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processWebflowRequest(WebflowExecutorImpl.java:546)
at com.bea.portal.appflow.PortalAppflowHelper.invokeWebflow(PortalAppflowHelper.java:166)
at com.bea.portal.appflow.PortalAppflowHelper.invokeWebflow(PortalAppflowHelper.java:139)
at com.bea.portal.appflow.PortalAppflowHelper.invokePortletWebflow(PortalAppflowHelper.java:216)
```

Figure 2: Exception with its exception features

The parser To transform an exception text blob into an entity of exception features, a parser had to be defined. Thus, the structure of the exception was further divided into:

- First line (containing class name and class message)
- Second line (containing exception type and exception message)
- Relevant stack trace
- Irrelevant stack trace
- 0..n Caused by:s (including Raised on:s)
  - Caused by row containing exception type and exception message
  - Relevant stack trace
  - Irrelevant stack trace

A state machine to parse the features was implemented with four states:

1. First row – Next row will be the first row
2. Second row – Next row will be the second row (or part of it)
3. Stack trace – Next part will be part of stack trace
4. Irrelevant stack trace – Next part is irrelevant stack trace, caused by row, or "end of exception"

Using the state machine, each line was parsed in regard to which state the parser was in. Two approaches were tested to parse the individual row: by tokenization and by regular expressions.

- Tokenizer
  - Description
    The tokenizer splits the row into entities (words) based on a set of characters used as delimiters. Using this method, the parser goes through entity by entity and at each it determines the meaning of the entity that follows it. E.g. if entity 3 is "<error>" entity 7 will be the exception type.
  - Advantages
    * Complex logic can be implemented to parse virtually any form of exception syntax.
  - Disadvantages
    * Difficult to update for new syntaxes – needs a recompilation and redeployment
    * Cumbersome to program
    * In general slow performance
    * Might yield unexpected results if unknown syntax is encountered.
- Regular expressions

- Description
  Regular expressions is an industry standard of describing patterns for string matching. It is made up of a pattern string, which is compiled[6] and then used to match text input. For more information see footnote. [7]
- Advantages
  * Standard and well known way of writing text matching.
  * In general a fast method (depending on the complexity of the pattern).
  * In contrary to the tokenizer, it will always return something even if the syntax is unknown. With unknown syntax, it is likely that an empty string is returned; which is far advantageous compared to thrown exceptions.
  * Easily updateable, only a pattern needs to change to incorporate new syntaxes.
- Disadvantages
  * Not possible to model everything into a matching expression.
  * Difficult to understand and maintain

Conclusions Exception syntax is very varying and inconsequent. Thus, it is very complex to write the parsing by a tokenizer. However, with the tokenizer method it is more possible to adapt to each and every case of the exception syntax. As time goes, new exception syntax will be discovered and the need of updating the parser will be quite high and occur quite often. Thus, regular expressions are superior since they are updateable without recompiling the code.

To handle the exception, it is stored in an entity representation that consists of the exception features in string form. The entities are implemented as a double linked list to be able to represent the "Caused by" structure.

3.3 Matching rules

What is called matching rules in this thesis, is the actual method on how the exception features and matching algorithms are combined and used to calculate if and how much two exceptions match each other. The problem is to conclude what features that are most important, how they should be combined, and how the "Caused by" structure should be dealt with. Does the "top" exception matter more than the middle or bottom ones? Should they be combined in some sort of weighted sum?

3.3.1 Methodology

The purpose of the project is to construct methodologies to be able to find matching exceptions that seem to be a bit unsimilar, when they should in fact be treated as equal. Hence, the superior matching rule will be found by testing different matching rule candidates against each other. However, performance is

[6]The possibility to compile the regular expression is not general, but is implemented in Java.
[7]http://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=347265226

also a factor to take into consideration and the best matching rule is the one that produces a good enough result within reasonable time (which in this case is around one second at tops for a single exception matching query).

3.3.2 Analysis

An exception consists of one main exception and zero or more “caused by” exceptions, each consisting of a number of features. The first thing that has to be determined is which of the exceptions in the chain that is the most relevant one (if not all) and what features that makes it so. A meeting with Raphael Kubler and Eric Durand yielded three different approaches to test: to only match the “top” exceptions, to only match the “bottom” exceptions (the last caused bys of every exception), or to match them both. The exceptions in the middle of the exception chain was regarded as irrelevant, based on the experience in exception investigation and solving of the production support team.

Benchmarking A small benchmarking application was created to test out the different matching rules. Three different fuzzy string matching rules were used to minimize the impact of the fuzzy string matching algorithm in the tests. The rules were set up to parse a big number of known exceptions and the execution time was measured, see Table 1

Matching of 12 test exceptions (same as in the unit test cases for the parser)

	Only top	Only bottom	Both top and bottom
Lazy Levenshtein	0.203	0.172	0.823
Small Levenshtein	0.307	0.359	0.989
Standard Levenshtein	0.588	0.625	1.656

Table 1: Match rule benchmarking

Conclusions are that “only top” and “only bottom” are essentially the same performance-wise. However, with further investigation of the results gotten from the three different rules; the production team concluded that only the rule that used both “top” and “bottom” exceptions produced good enough results. So the matching rule that was finally chosen was the one that matched both “top” and “bottom” exceptions, despite it being the slowest one.

3.4 Algorithms

An exception’s features are in the majority of cases made up by strings and therefore algorithms for comparing strings must be implemented. The requirements are that strings should be measured for similarity, without the algorithm using too much time and resources.

3.4.1 Method

To find the best suited algorithm, the field of fuzzy string matching algorithms was investigated. What is interesting is how the special properties of each algorithm suites the particular circumstances of Java exception matching. The found candidates were implemented and benchmarked to determine which one that was the best suited for the task.

3.4.2 Analysis

After studying the field of string matching on Wikipedia and Topcoder[8] a number of properties were concluded. String matching is generally done through calculating the so called editing distance between the strings. The editing distance is defined as the number of single operations (such as inserting a character, deleting a character, transposing two characters etc.) needed to transform the first string into the second one. Thus, equal strings have an editing distance of 0 and the more dissimilar the strings are – the higher editing distance they have.

Examined algorithms

1. Levenshtein distance[9]

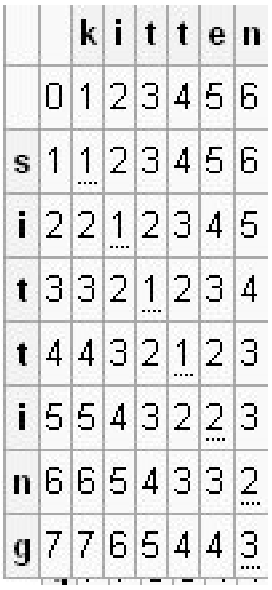
   (a) General description
   The Levenshtein distance is a measure of the minimum amount of edits, in the form of inserted characters, changed characters, or deleted characters, that are needed to transform one string into another.
   Good example taken from the Wikipedia article[10] on Levenshtein distance:
   “As an example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other and there is no way to do it with fewer than three edits:
   1.kitten → sitten (substitution of 's' for 'k')
   2.sitten → sittin (substitution of 'i' for 'e')
   3.sittin → sitting (insert 'g' at the end). “

   The standard way of calculating the Levenshtein distance between two strings is by using dynamic programming;

   The matrix is initialized with the two strings as its sides and a $[0..n]$, $[0..m]$ cost count in the first row and column, where n and m are the respective string lengths.
   The matrix is traversed from upper left to lower right with the cost rules for each position in the strings, defined as:

      i. If the characters are equal; the cost is the same as the one upper left from the cost being calculated. Thus, if the strings are completely equal, the cost will be 0 since it will propagate from upper left to bottom right in a diagonal.
      ii. If the characters are not equal, the cost is the minimum of the one to the left +1, the one to the upper left+1, and the one above+1. In effect, this constitutes the insertion, substitution, and deletion of a characters.
      iii. When the lower right corner is reached, the total string editing distance is the cost of that cell.

[8] http://software.topcoder.com/catalog/document?id=8457494
[9] http://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=222332508
[10] http://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=222332508



		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Figure 3: Dynamic programming Levenshtein example

As a side note, the exact steps to transform the first string into the other one is possible to store by storing the routes taken in the matrix and then backtrack when all the cost calculations are done. However, this is irrelevant for the particular application of Java exception matching.

(b) Applicability to exception matching
The Levenshtein algorithm is highly applicable to exception matching since exceptions can be matched based on the similarity of their features – which are in fact strings.

(c) Time/space complexity
Given the dynamic programming approach, the time and space complexity is O$[n \times m]$, with n and m being the respective lengths of the input strings

2. Damerau-Levenshtein [11]

(a) General description
The Damerau-Levenshtein is a modification to the original Levenshtein algorithm with the addition of another editing action: transpositions. Hence, it measures the distance between strings based on insertions, substitutions, deletions and transpositions

(b) Applicability to exception matching
The added benefit compared to the original Levenshtein is mostly applicable in problems concerning strings that are misspelled. Exceptions are computer generated and thus the edit action of transpositions adds no gain. This algorithm is therefore discarded in favor of the original Levenshtein.

[11] http://en.wikipedia.org/w/index.php?title=Damerau%E2%80%93Levenshtein_distance&oldid=218621877

3. Needleman-Wunsch[12]

   (a) General description
   The algorithm is used to compute the optimal alignment between two similar strings and the metric is basically a gap penalty. If the strings are identical, there will be no gap in the alignment of them and the number of gaps will increase proportionally with the dissimilarity of the strings. The calculations are implemented using a dynamic programming approach.

   (b) Applicability to exception matching
   The alignment itself is of no interest to exception matching, however the gap penalty can very much be used as a quantification of how similar exceptions (strings) are. However, it has no gain over the standard Levenshtein distance since they both are implemented through dynamic programming and both have the same time/space complexity

4. Jaro-Winkler [13]

   (a) General description
   The Jaro-Winkler algorithm also matches the strings through dynamic programming.
   Each string is looped through, character by character. If the current character in one string exists in the other string within the distance of half the string length, it is considered a match. The total editing distance between the strings is a normalized average of individual character matches. Therefore, the algorithm does not take character order into consideration and slightly dissimilar strings can thus be considered a perfect match. The algorithm is designed to generate a normalized result, regardless of the strings compared, the Jaro-Winkler distance is always between 0.0 and 1.0.

   (b) Applicability to exception matching
   Since the characters in different exceptions are highly unlikely to be permutations of each other, the Jaro-Winkler distance might be used as a similarity measurement of exceptions. It is not as intuitive as the Levenshtein distance, but the metric makes perfect sense for the sole purpose of getting a match percentage between exceptions. It does not offer any improvement to Levenshtein though, since the time/space complexity is the same.

   (c) Time/space complexity
   All characters in string 1 is compared one by one with half the characters of string 2 and then vice versa. This gives $\Theta[2 \times (\frac{n \times m}{2})] = \mathrm{O}[n \times m]$.

5. Bitap [14]

[12] http://en.wikipedia.org/w/index.php?title=Needleman%E2%80%93Wunsch_algorithm&oldid=215012856
[13] http://en.wikipedia.org/w/index.php?title=Jaro-Winkler_distance&oldid=216235867
[14] http://en.wikipedia.org/w/index.php?title=Bitap_algorithm&oldid=211541671

   (a) General description
   The Bitap algorithm is used to search for patterns in a text using mainly bit wise operations and calculates the Levenshtein editing distance between them. Although the time complexity is O$[n \times m]$, the algorithm still has better performance than the Levenshtein algorithm (which also performs in O$[n \times m]$), since bit wise operations are very fast.
   (b) Applicability to exception matching The bitap algorithm only performs good with patterns of small sizes – such as the word length of the machine. Since the pattern in this case is one of the exceptions (which is considerable longer than the word length), the algorithm is not suitable for exception matching.
   (c) Time/space complexity O$[n \times m]$

6. Boyer-Moore [15]

   (a) General description
   The Boyer-Moore is a fast string searching algorithm. It finds exact matches of a string within another string and actually performs better the longer the strings are.
   (b) Applicability to exception matching
   In string matching terms speaking, exceptions are long. Therefore, the fact that the algorithm performs better on long strings than short is a good fact. The time/space complexity is also really good. Unfortunately, it is completely useless for Java exception matching since it only find exact matches.
   (c) Time/space complexity
   Worst case: O$[n]$

7. Ukkonen [16]

   (a) General description
   The algorithm compares the strings with a threshold value that varies from 0 to the editing distance between the strings and stops as soon as the correct editing distance is found. Thus the worst case is the length of the longest string, since it is the maximum editing distance[17]. This algorithm was quite hard to find good information on, but it needed not to be investigated further since another algorithm has the same complexity – see the "Tweaking and specialization" section.
   (b) Applicability to exception matching
   The algorithm is just as suited to exception matching as the original Levenshtein distance algorithm.

[15]http://en.wikipedia.org/w/index.php?title=Boyer%E2%80%93Moore_string_search_algorithm&oldid=221497526

[16]http://software.topcoder.com/catalog/document?id=8457494

[17]To transform string A (shortest) into string B(longest), one way is to substitute all the characters in A with the Length(A) first characters in B and add the rest of the characters in B. Thus the editing distance will always be less or equal to the length of the longest string in the comparison.

(c) Time/space complexity
O$[n \times d]$ where n is the length of the smaller of the two strings and d is the distance between them. Thus, the time complexity is between O$[n]$ and O$[n \times m]$ since the worst case distance is the length of the longest string.

Tweaking and specialization of the algorithms

- Algorithm variations
  - *Thresholding*
    For Java exception matching, the exact editing distance is not necessarily relevant. If it is big, the exception just is not a match. Hence, the performance and thus the running time of the algorithms can be improved by stopping the calculations after a certain distance is found – the threshold. If the threshold is reached, then the strings are considered to be 100% dissimilar and the program can go on to the next calculation instead.
  - *Lazy evaluating Levenshtein*
    In the investigation of the Levenshtein algorithm, a tweak of it was encountered. It is possible to rewrite it to utilize lazy evaluation. Lazy evaluation has the effect that only the values actually needed to reach the result is evaluated and execution time can be reduced. This reduces the time complexity into O$[m[1 + d]]$ where m is the length of the longest string and d is the distance between them.
    The principle of the algorithm is simple; the Levenshtein algorithm uses dynamic programming to reach its result and thus the resulting editing distance between the strings will end up in the lower right corner of the distance matrix. Using regular Levenshtein, the distance matrix will be calculated from left to right, top to bottom. A much faster way to reach the bottom right is to find the diagonal from upper left to bottom right which contains the lower right element and then traverse the matrix diagonally instead of horizontally. However, to calculate a value in the matrix, one must still know the three elements surrounding it on the top, top-left, and left. So in the lazy evaluating diagonal algorithm, these values has to be calculated first and thus by only using this modification we gain nothing (since the whole matrix has to be calculated nonetheless).
    But an observation[18] is made that when the minimum cost function is evaluated, not all values need to be calculated. The minimum cost function takes the minimal of three values – top, top-left, and left) and returns the smallest of the three. Summarized; given that the element on top is less than the element in top left; then the element on the left is greater than or equal to the top left value. Hence, the left value doesn't have to be calculated at all. If a value which would result of a whole new diagonal to be calculated can be ignored, a huge gain in performance is achieved. This results in a performance complexity of O$[n]$ $\leq$O$[x]$ $\leq$O$[n \times m]$. Hence, the algorithm runs in O$[n]$

[18]http://www.csse.monash.edu.au/~lloyd/tildeStrings/Alignment/92.IPL.html

if the strings are equal and it runs in O$[n \times m]$ if they are completely different. This matches the complexity of the original Levenshtein algorithm with highly probable performance gain in reality. Added the benefit of thresholding the algorithm; the worst case complexity will never be reached since the calculations will be aborted beforehand. Thus, the lazy evaluating Levenshtein with thresholding will always be faster and smaller in space, than the original one. At least in theory.
One note to mention; the Lazy algorithm needs to be implemented using Objects in the Java implementation. The original Levenshtein can be implemented using nothing but standard integer arrays as memory storage. For small strings the original may run faster than the lazy one, given the overhead needed in time and space for creating Java Objects.

– *Small memory Levenshtein algorithm*
Another variation of the standard Levenshtein algorithm concerns the memory requirements. Given two large strings, the memory requirement to store the distance matrix grows very big; O$[n \times m]$ and it is not uncommon for the application to run out of memory. In exception matching in particular, the stack trace of an exception can be quite long and OutOfMemory is a reasonable outcome. One common solution to this is generally implemented based on the fact that to calculate a value in the distance matrix, only the current row and the one above it is needed. Hence, just store those two rows at all times. This obviously reduces the space requirements from O$[n \times m]$ to O$[n \times 2]$), where n is the length of the *smallest* of the two strings. The performance is still however the same as the original Levenshtein algorithm, since all values need to be calculated and unfortunately it cannot be combined with the lazy evaluating algorithm. In reality, the time to execute the algorithm may be smaller than in the original algorithm since there is an overhead in time to create the large chunks of memory needed to store the distance matrix.

Benchmarking To find the best performing algorithm for the application of Java Exception matching, a benchmarking tool was created. The tool compared a number of long and short strings of varying complexity and measured the running time and space requirements.

Definition of terms used in the benchmarking

Equal strings Two random strings that are equal

Similar strings Two random strings that are up to 30% different. (close or equal in length)

Random strings Two completely random strings (also in length) short string: Between 20 and 100 characters long string: between 1000 and 3000 characters

What is wanted from this benchmarking is an estimate of how the algorithms work in the extreme cases: equal, and totally random strings. But also how it works on average in between, under circumstances which resembles exceptions somewhat. Hence, the similar strings tests were added.

Java exceptions are very structured and the text is quite repetitive and thus they are quite far from random strings. This property has been simulated somewhat by limiting the alphabet out of which the strings are created from. All strings are made up from solely the lowercase letters a-o and thus many equalities and repetitions will occur.

The similar strings are constructed as follows: A string out of the limited alphabet is constructed randomly with a random length. This string is duplicated and the copy is then modified in three ways: adding, deleting and changing of characters. A random number of up to 10% of the the length of the string is generated for each operation and that number of modifications are then made. Finally the two strings are used for matching each other.

The selections of boundaries of the string lengths and also the construction method and degree of similarity of the strings has not been investigated further since the values seem to have given a good estimate on the behaviors of the different algorithms. It is not the actual numbers, but the algorithm behaviors that are interesting.

100 Strings generated, Threshold: drop at 40% mismatch

	Lazy Thresholded	Lazy Leven	Apache	Standard	SmallStandard	SmallStandard Threshold
Max size	5500	4300	4000	4000	50000+	50000+
Equal						
short	0.0	0.0	0.016	0.016	0.015	0.016
long	0.015	0.0	16.236	15.282	12.235	12.25
Similar						
short	0.016	0.0	0.016	0.015	0.0	0.016
long	6.656	6.516	18.017	16.704	13.86	13.829
Random						
short	0.016	0.015	0.0	0.016	0.0	0.016
long	10.454	25.439	15.214	14.007	11.226	8.713

Table 2: Fuzzy string matching benchmarking (time in seconds)

Table conclusions

- *Space requirements*
  The small memory version of the algorithm clearly outperforms the others. In fact, I have not found any limit to what sizes it can handle, but given the context of exception matching; a size of 50000 characters is more than sufficient. However, the others are quite likely to run out of memory during exception comparison every once in a while given their max length of approximately 5000 characters.

- *Performance*
  The string composition of exceptions can mostly be seen as either long

and similar or long and completely dissimilar with an addition of a couple of short string features. It is the length that is dominant in the calculation time. Thus, the winner should be the Thresholded Lazy evaluating Levenshtein, with the Small Standard Thresholded as a runner up. A note on the running time of long random strings and the small standard thresholded algorithm has to be made. At first glance, one may think that the running time should be approximately the same on all long strings since it is a dynamic programming algorithm. However, the shorter time on long random strings is probably due to the fact that the threshold gets hit faster than in the other cases.

3.5 Program structure

3.5.1 Methodology

What is meant by program structure in this report, is how the system is architectured to implement all parts needed to do Java exception comparisons. This includes what language, what models, what patterns used, and so forth. To find out what combination that is the best suited one a number of parameters was analyzed: the context in which the program will operate in, what users will use it and how, for how long will it be used, what are the needs of updateability and maintainability, what are the need of extendability, and so forth. To investigate this, first of all the project description was consulted. Practices learned during my education and prior experience was considered and information on the Internet was analyzed. Another obvious big source of knowledge was the team in which I worked and the team leader Raphael Kubler.

3.5.2 Analysis

From the project description (found in the introduction chapter 2.3) a few constraints can be extracted: the application will interact with external systems, use a database, have a graphical user interface, and also a web service entry point. From the production support team it was learned that the project was mainly about Java exceptions, the standard development language used is Java, and many of the servers where running Java EE applications. Thus Java would be a quite convenient selection of language for the program. For the user interface, there is mainly two ways to go: old fashioned windows application interface or a purely web based interface. This far in the process, none is really said about what would be best for the main user interface other a web service is desired in addition to it.

There is however not anything that limits the application to use only one of the options, so to be able to implement both versions, a variation of the model view controller structure was decided upon. This way the main application logic is interface independent and both a windows interface and a purely web based interface can be built on top of it. An additional layer was added "vertically", a utility layer, to keep things better structured logically. The utility layer contained classes that all layers needed access to, such as data entity carriers and, common functions.

The layers are dependent on each other from top to bottom, the view can only access the controller layer and the controller layer can only access the

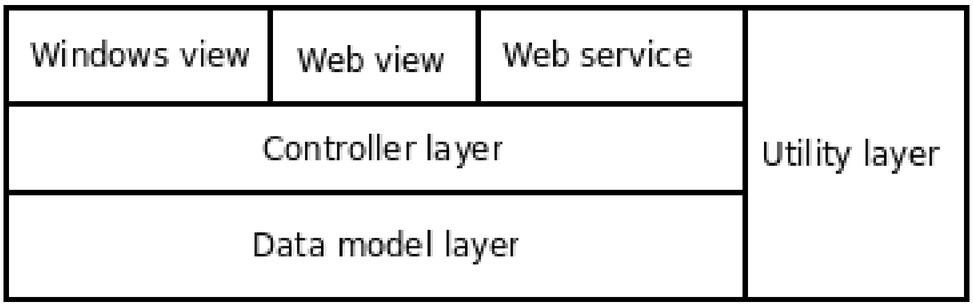


Figure 4: Architecture of solution

data model layer, but not in the reversed direction. Hence, the data model implementation can be completely replaced (e.g. if the underlying database brand is changed) as long as it implements the same interface as the previous model. The same obviously goes for the controller layer. This way there can of course be as many user interfaces connected to the controller layer as needed, all independent of each other. The only exception is the utility layer, which is accessible by all other layers but cannot access any other layers itself.

Another great advantage of this model is that the program structure is easy to maintain, understand, and also easy to extend and build further upon. The diagrams of the final layers are found in 4.4 and the design of the web view in particular, is found in 4.6.

3.6 Storage (database) model

To be able to match the Java exceptions, some sort of storage of known exceptions are needed to match against. In the construction of a good Java exception matching method for matching in real time, the design of the data storage is a highly important part.

In this particular implementation, the matched exceptions should be paired with the Problem tracking record (PTR) that Amadeus keeps in order to resolve the issue. The PTRs are not searchable and thus the PTR association must make use of the storage of known exceptions kept by the application. A standard and definitely fully good solution is to use a modern DBMS. Amadeus is using MS SQL Servers and for this project a database was created in one of them. The question to investigate is thus how the information should be structured into tables and relations. Also what data and how much data, but that is more a general question of the project than just the database.

3.6.1 Methodology

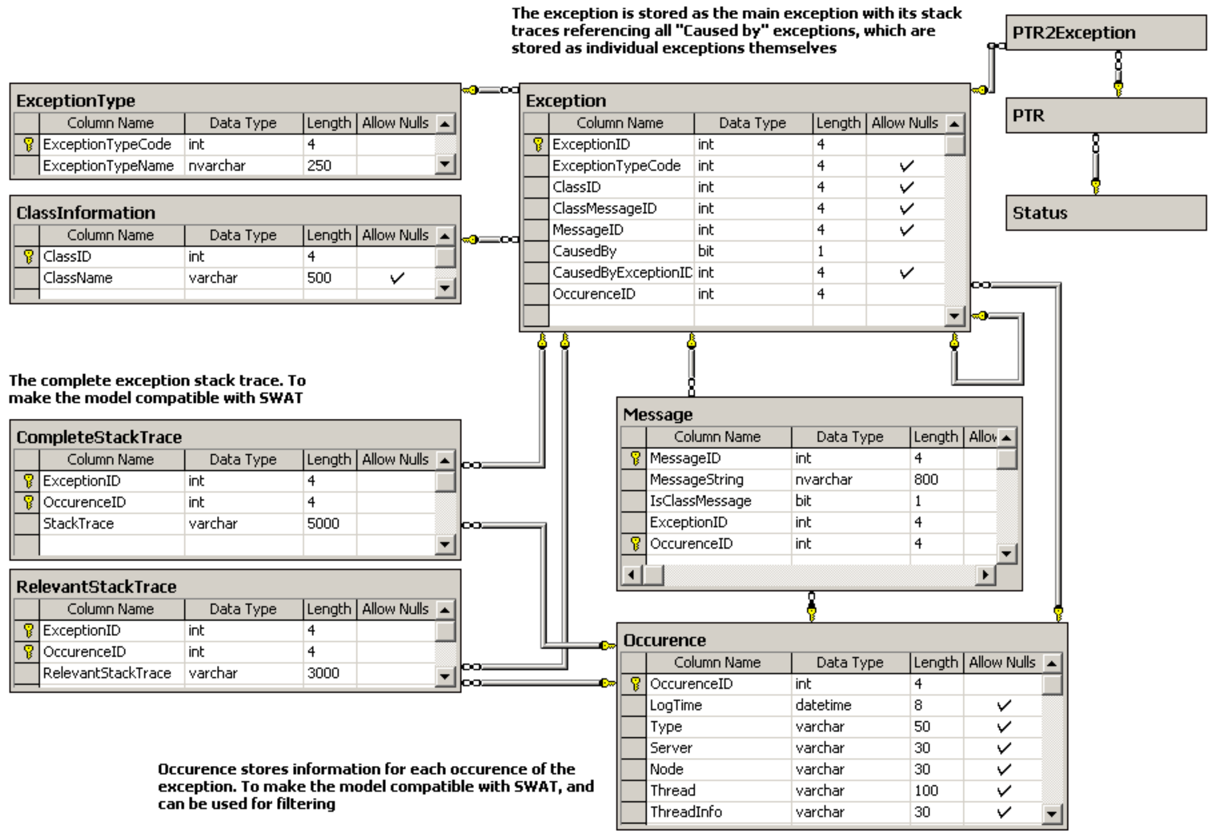
The problem was analyzed and when a model was concluded, database diagrams were proposed. After some rounds of feedback from NRE and PSU teams a final database diagram was set.

When implementing a database, a set of methodologies exist to ensure things as data integrity, security, and so forth. Amadeus enforces a number of them, which were used and in addition to that some were investigated and implemented as well.

3.6.2 Analysis

A first suggestion The first suggestion is a normalized data model, which captures all the selected features of the exception. It also contains tables to store the occurrences and complete stack traces of the exceptions, in order to be able to have all available information stored.

Taking SWAT and SPIN into consideration After a meeting with NRE and PSU it was decided that this project should use the internal framework SPIN and also be compatible with the internal SWAT project. Hence, the database model was to follow the SPIN policies and work together with SWAT. The model was now to be implemented within the SWAT database with the occurrence and complete stack trace tables taken care of SWAT. The model (tables) specific to this application would handle just one copy of each exception with a shortened "relevant stack trace" and only by combining it with the SWAT tables, a complete exception instance could be recreated. However, to build the link between the SWAT tables and the Exception Tracker tables, the Exception tracker obviously has to be run. See Figure 6



The exception is stored as the main exception with its stack traces referencing all "Caused by" exceptions, which are stored as individual exceptions themselves

ExceptionType

Column Name	Data Type	Length	Allow Nulls
ExceptionTypeCode	int	4	
ExceptionTypeName	nvarchar	250	

ClassInformation

Column Name	Data Type	Length	Allow Nulls
ClassID	int	4	
ClassName	varchar	500	✓

Exception

Column Name	Data Type	Length	Allow Nulls
ExceptionID	int	4	
ExceptionTypeCode	int	4	✓
ClassID	int	4	✓
ClassMessageID	int	4	✓
MessageID	int	4	✓
CausedBy	bit	1	
CausedByExceptionID	int	4	✓
OccurenceID	int	4	

The complete exception stack trace. To make the model compatible with SWAT

CompleteStackTrace

Column Name	Data Type	Length	Allow Nulls
ExceptionID	int	4	
OccurenceID	int	4	
StackTrace	varchar	5000	

Message

Column Name	Data Type	Length	Allow
MessageID	int	4	
MessageString	nvarchar	800	
IsClassMessage	bit	1	
ExceptionID	int	4	
OccurenceID	int	4	

RelevantStackTrace

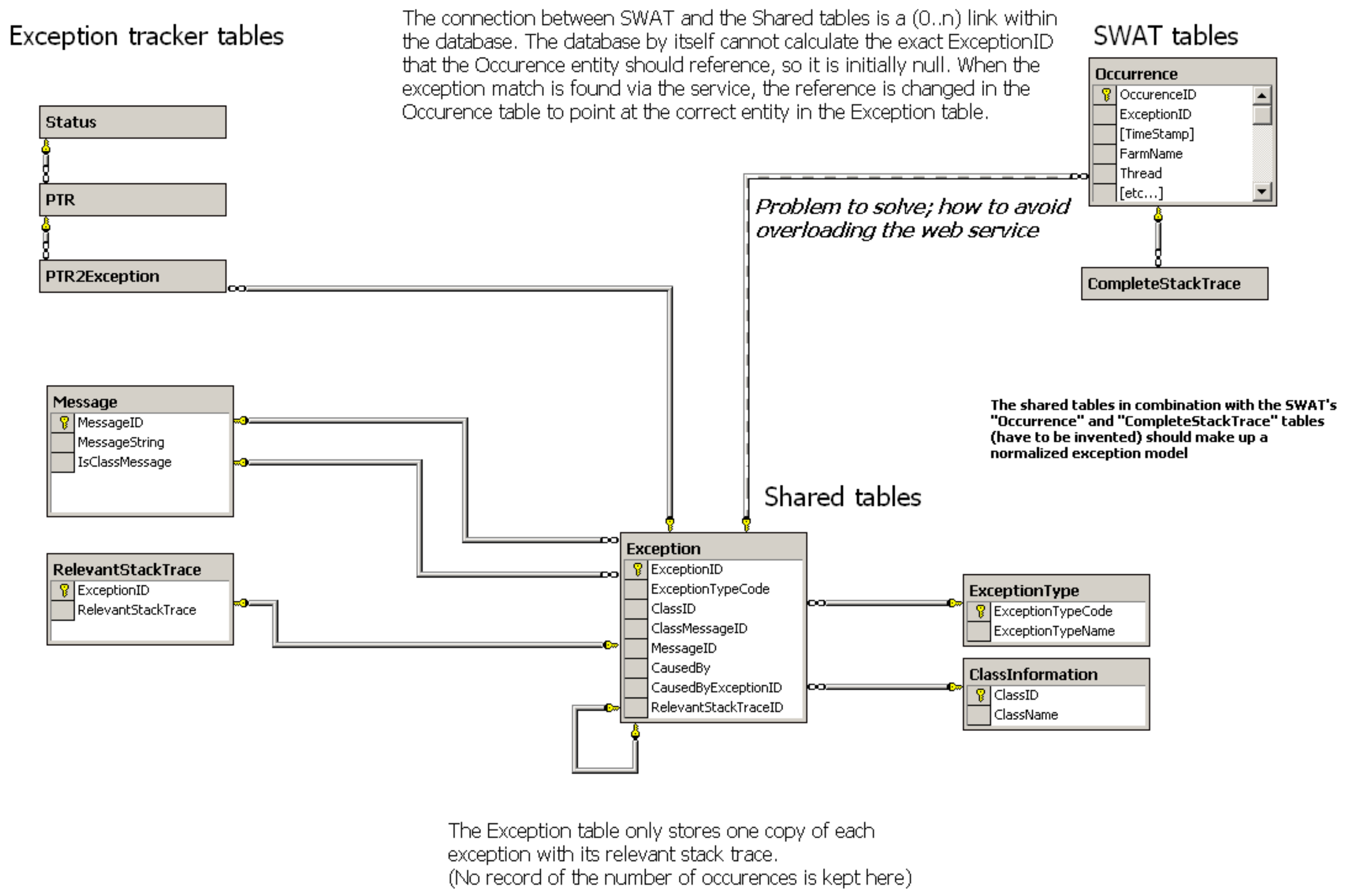
Column Name	Data Type	Length	Allow Nulls
ExceptionID	int	4	
OccurenceID	int	4	
RelevantStackTrace	varchar	3000	

Occurence

Column Name	Data Type	Length	Allow Nulls
OccurenceID	int	4	
LogTime	datetime	8	✓
Type	varchar	50	✓
Server	varchar	30	✓
Node	varchar	30	✓
Thread	varchar	100	✓
ThreadInfo	varchar	30	✓

Occurence stores information for each occurence of the exception. To make the model compatible with SWAT, and can be used for filtering

Figure 5: Normalized data model



The connection between SWAT and the Shared tables is a (0..n) link within the database. The database by itself cannot calculate the exact ExceptionID that the Occurence entity should reference, so it is initially null. When the exception match is found via the service, the reference is changed in the Occurence table to point at the correct entity in the Exception table.

Problem to solve; how to avoid overloading the web service

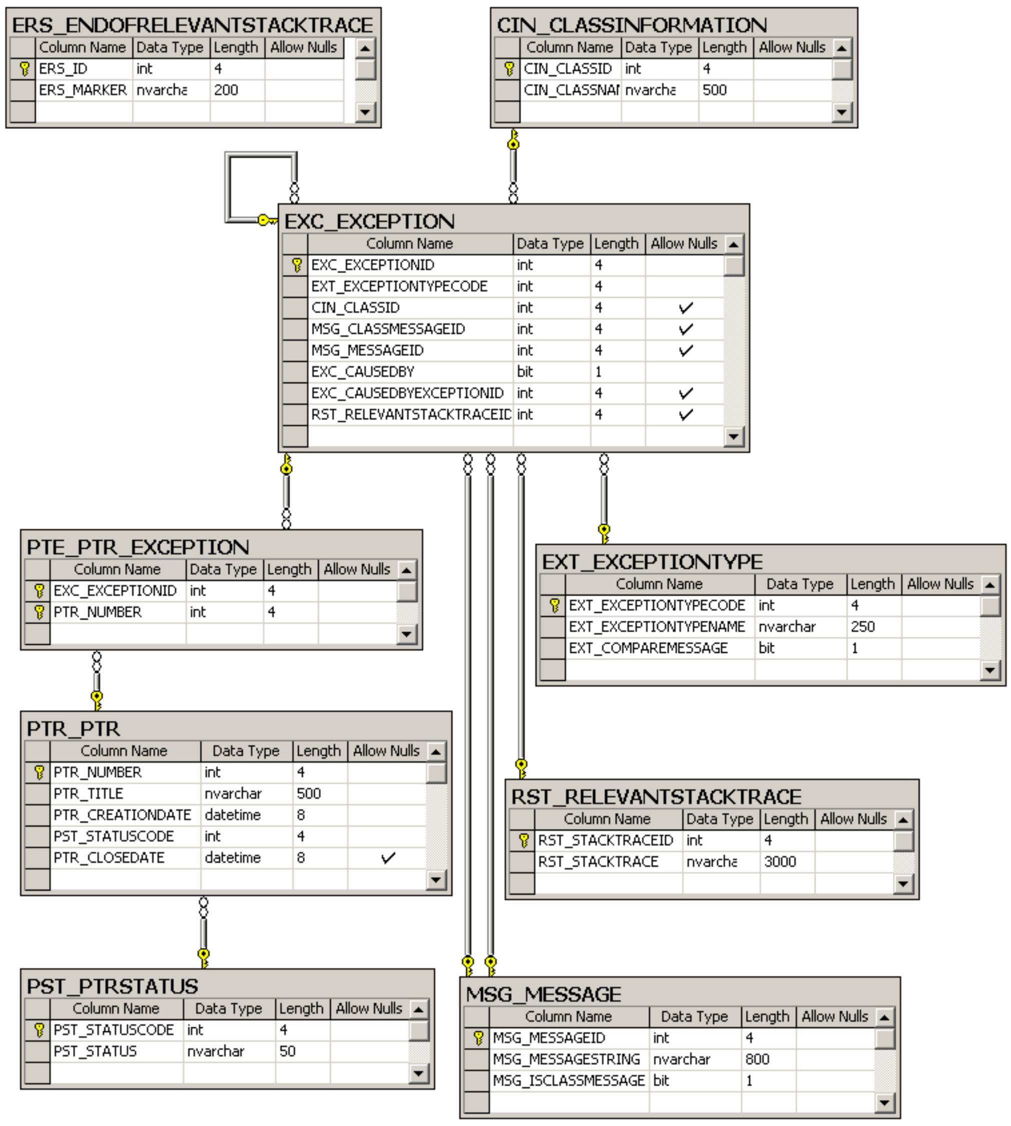
The shared tables in combination with the SWAT's "Occurrence" and "CompleteStackTrace" tables (have to be invented) should make up a normalized exception model

The Exception table only stores one copy of each exception with its relevant stack trace.
(No record of the number of occurences is kept here)

Figure 6: Data model in conjunction with S.W.A.T

Limit to internship subject → removing occurrences After a meeting with Raphael Kubler, the database model was revised to more closely match the original internship description, i.e. the occurrences and complete stacktraces was removed. In reality, the effect was to move the Exception Tracker database out of the SWAT database to be a completely stand alone database and move all of what was necessary to recreate complete exception occurrences to the responsibility of the SWAT project.

The data model is only supposed to store the data which is relevant for matching with found exceptions. In practice, this means that only the first occurrence of an exception is stored – only unique exceptions are interesting – and the number of occurrences is not stored at all. Since the exception is not stored in full, it is not possible to reconstruct all the information given in the original exception. However in combination with SWAT, this will be possible. See Figure 7



ERS_ENDOFRELEVANTSTACKTRACE

Column Name	Data Type	Length	Allow Nulls
ERS_ID	int	4	
ERS_MARKER	nvarcha	200	

CIN_CLASSINFORMATION

Column Name	Data Type	Length	Allow Nulls
CIN_CLASSID	int	4	
CIN_CLASSNAI	nvarcha	500	

EXC_EXCEPTION

Column Name	Data Type	Length	Allow Nulls
EXC_EXCEPTIONID	int	4	
EXT_EXCEPTIONTYPECODE	int	4	
CIN_CLASSID	int	4	✓
MSG_CLASSMESSAGEID	int	4	✓
MSG_MESSAGEID	int	4	✓
EXC_CAUSEDBY	bit	1	
EXC_CAUSEDBYEXCEPTIONID	int	4	✓
RST_RELEVANTSTACKTRACEIC	int	4	✓

PTE_PTR_EXCEPTION

Column Name	Data Type	Length	Allow Nulls
EXC_EXCEPTIONID	int	4	
PTR_NUMBER	int	4	

EXT_EXCEPTIONTYPE

Column Name	Data Type	Length	Allow Nulls
EXT_EXCEPTIONTYPECODE	int	4	
EXT_EXCEPTIONTYPENAME	nvarchar	250	
EXT_COMPAREMESSAGE	bit	1	

PTR_PTR

Column Name	Data Type	Length	Allow Nulls
PTR_NUMBER	int	4	
PTR_TITLE	nvarchar	500	
PTR_CREATIONDATE	datetime	8	
PST_STATUSCODE	int	4	
PTR_CLOSEDATE	datetime	8	✓

RST_RELEVANTSTACKTRACE

Column Name	Data Type	Length	Allow Nulls
RST_STACKTRACEID	int	4	
RST_STACKTRACE	nvarcha	3000	

PST_PTRSTATUS

Column Name	Data Type	Length	Allow Nulls
PST_STATUSCODE	int	4	
PST_STATUS	nvarchar	50	

MSG_MESSAGE

Column Name	Data Type	Length	Allow Nulls
MSG_MESSAGEID	int	4	
MSG_MESSAGESTRING	nvarchar	800	
MSG_ISCLASSMESSAGE	bit	1	

Figure 7: Final database diagram

Relevant information When exceptions are compared, performance is an important issue. Therefore the amount of information must be limited so only what is truly relevant is used in the computations. Since fuzzy string matching is very costly, the more filtering of exact matching parts that can be done by the database, the better performance. In the model of an exception, the information has been divided into what can be matched exactly and what needs to be matched by fuzzy matching. See Figure 8

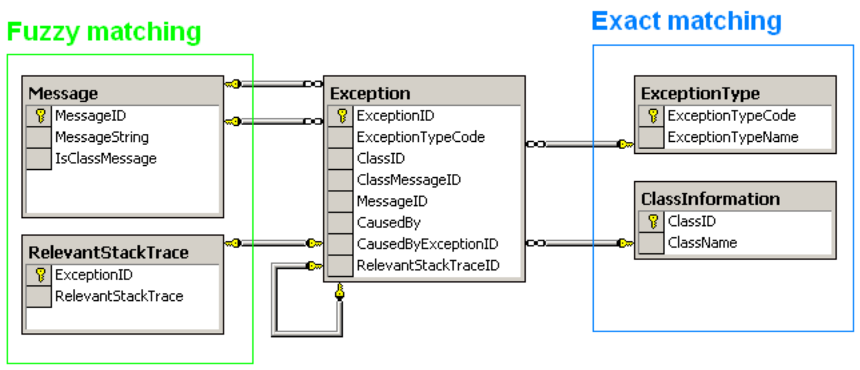


Figure 8: Fuzzy vs exact matching in database

The fuzzy matching is a very costly operation and therefore it must be fully optimized and avoided in all cases where it is not necessary. As a part of the optimization, the stack trace of the exception is limited to what is really necessary for the comparison. What constitutes a "relevant" stack trace then? There isn't really much research done in this area from what I could find, so it is very much a matter of empirical research. At a first glance, the stack trace can be divided into parts based on "caused by" clauses. Each "caused by" clause and of course the main exception itself, has each its own stack trace. Further, the stack trace can be divided into application specific stack trace and other stack trace. The application specific stack trace stems from the application itself, while the other stack trace comes from the platform that is executing the application – in this case WebLogic. Only the application stack trace is relevant in this project, since it's in the application that the exception is thrown and should be resolved. Thus it is the application stack trace that is defined as the relevant stack trace.

General database implementation details There are essentially two ways to go when manipulating data, either to allow dynamic SQL to run on the database or to only allow the use of stored procedures. By allowing dynamic SQL, the database is (or can be) updated by writing SQL queries directly in the application that is used along with the database. On the other hand, if only stored procedures are allowed, all SQL is written within the database and the application only sends parameters when data manipulation is to be done in any way. There are mainly two advantages with using the latter approach, stored procedures. First of all the performance is better since MS SQL Server precompiles all stored procedures. Secondly, it is easier to fine tune security permissions of the applications, i.e. the application can update data using the update stored procedure without actually having write-permissions to the underlying table, as long as it has execute permission on the stored procedure. And of course it is limited to running only predefined SQL queries (those in the stored procedures) and cannot therefore run potentially harmful queries that are crafted by the malicious user. The disadvantages are that it takes time to write all possibly needed stored procedures and also that the application is later limited to running the stored procedures and cannot use any specially crafted SQL queries which could possibly yield better performance and/or easier programming.

For data insertion and manipulation, the way of doing everything through stored procedures chosen. This means that insert, update, delete and get procedures corresponding to every table was created. Since this is a cumbersome and quite time costly job, a a tool called MyGeneration[19] was used, which generates stored procedure code for each table based on a template.

Data integrity In a normalized database, data integrity is of importance. More concretely no duplicate data should exist and when entries in one table references an entry in another table, the other entry must actually exist. Problems with this can arise during data insertion, e.g. some data insertion can fail during data modification and therefore a reference is left hanging loose. The responsibility to prevent breaks in data integrity should reside as much as possible within the database itself – it should not be possible to manipulate data in a way which breaks the data integrity. Although, it is not always possible to achieve perfect data integrity checks using SQL alone, so additional logic might be needed in the application using the database. Measures to uphold data integrity is the use of constraints, error handling, transactions, triggers and in a way; testing.

Constraints Constraints in a database can be put in place for ensuring that only "valid" values can be stored by regarding some set of rules. The advantage by implementing constraints is that one cannot later have an invalid value stored due to programming errors, since the database itself just won't allow it. In this data model I'm enforcing unique, referential and not null constraints.

Unique constraints make sure that no two or more columns in the database table can contain the same value. However, these are implemented as indexes and MS SQL Server has a limitation on the size of indexes, which leads to that fields of size bigger than 900 bytes (such as RST_STACKTRACE) cannot have unique constraints. So to avoid being inconsistent, no unique constraints are added, but instead the insert and update stored procedures checks that they are not inserting any duplicates. One exception for the unique constraints is that all primary keys automatically have a unique constraint on them.

Referential constraints (foreign key constraints) make sure that when a column in table A references a value in table B, the value must exist. There is foreign key constraints added between all tables that references each other in the data model and the EXC_EXCEPTION also has a foreign key constraint on itself to be able to chain "caused by exceptions". In the EXC_EXCEPTION table nulls are allowed in the self reference since not all exception is caused by other exceptions.

Not null constraints decide if a field may contain NULL values or not. This makes sure that it's not possible to add null data to fields which should contain values and it is therefore a good practice to enforce NOT NULL where appropriate. This practice is followed in the data model. However, since exceptions can look so different, the EXC_EXCEPTION table actually allows nulls in most fields.

Check constraints are used to make sure that values entered are conforming to special rules, e.g. an integer should only be able to take on certain values or a date must be later than another date etc. This is only needed at one place in the

[19]http://www.mygenerationsoftware.com

data model, in the PTR_PTR table to make sure that the PTR_CLOSEDATE is greater or equal to (or null) the creation date .

Error handling As in all coding, running SQL queries can result in errors. In MS SQL Server this is indicated by a value different from 0 in the internal @@error variable, which is set after each statement that is executed. Since it is set after each statement, the error checking code becomes quite lengthy, for example :

```
IF (@@error <> 0) print @@error
```

will always print out 0, since it is highly unlikely that the IF statement will fail. This is handled by assigning the value of @@error to a local variable and then acting upon that value.

```
DECLARE @err int
SELECT @err = @@error
IF @err <> 0
BEGIN
    print 'ERROR: ' + cast(@err as nvarchar)
END
```

When a stored procedure is executed, it is a bit more complicated since an error can arise both from a failure within the stored procedure as well as from the failure of the execution of it. Hence, the @@error must be checked directly after the call as well as the returned value from the stored procedure:

```
DECLARE @err int
EXEC @err = storedProcedure @Variable1=@Value1
SELECT @err = coalesce(nullif(@err, 0), @@error)
IF @err <> 0

    BEGIN print 'ERROR: ' + cast(@outerERR as nvarchar)
END
```

Also, the stored procedure should notify its caller if an error occurs within it that it does not handle itself, by setting its return value to something else than 0. This is unfortunately not possible in the "Get procedures" which returns a dataset, but on the other hand it is highly unlikely that they will fail since they more or less is simple "SELECT statements".

Transactions If an error occurs, all changes up to that point should be reverted to avoid semi complete data (in this case exceptions) stored. This atomicity is accomplished by using transactions. A transaction can be either committed or rolled back, which means that all changes within the scope of the transaction are either saved in the database or none are. The stored procedures for inserting and deleting an exception relies heavily on this since an exception references a lot of parts stored in other tables. If some part fails to be stored when inserting an exception, the remaining parts of the incomplete exception should not be inserted either. As well as when an exception is removed and the clean up of its referenced parts fails, the deletion should be aborted.

If an executed stored procedure fails within the scope of a transaction, the transaction will not treat this as an automatic rollback. Therefore the error must be detected and the transaction rollback must be explicitly called. The complete error handling code for each stored procedure call is therefore implemented as:

```
DECLARE @err int
EXEC @err = storedProcedure @Variable1=@Value1
SELECT @err = coalesce(nullif(@err, 0), @@error)
IF @err <> 0
BEGIN

    print 'ERROR: ' + cast(@outerERR as nvarchar)
    ROLLBACK TRANSACTION

END
```

Within the insertion and update stored procedures, a check is done to ensure that the insertion or update will not result in any duplicate data. Since the check for duplicates and modification of data is two distinct steps, they have to be performed as one atomic action if the duplicate check should be reliable when the data is inserted. Hence, these stored procedures contain transactions and they become nested when they are executed by the spEXC_I_INSERT procedure. One problem with nested transactions in Transact SQL is that the inner transaction cannot be rolled back separately from the outer. Luckily, this poses no problems in this particular case since no data modifications are made when the procedure checks for duplicates. It is therefore safe (data integrity-wise) to commit the transaction instead of rolling it back in the case of a duplicate found.

Tests In regular programming, automatic tests are often included in the process. This makes sure that the code does what it is supposed to and is used to verify that everything still works later on when changes have been made (a.k.a. regression tests).

The application uses automatic tests via JUnit and since the stored procedures are mostly automatically generated, it would be a good idea to implement automatic testing of the database as well. The tests are written in SQL and then generalized and generated for all tables using MyGeneration. The good thing with this is that it results in pure SQL code, so no extra software is needed to execute the tests. All tests are run within a transaction, which is rolled back when the test is finished so that the tests do not leave any data behind when they are done. The transaction is also rolled back when some part fails, which has the effect that only the first potential error is found per test run. On the other hand this also makes sure that no tests fail due to previous errors when they really should not.

Problems with the auto generation method The EXC_EXCEPTION table is too complex for simple insert, update, retrieve, delete tests due to its heavy dependence on other tables, so tests of EXC_EXCEPTION had to be written manually.

Since the PTE_PTR_EXCEPTION table consists merely of primary key references (it is a many to many link table), it could not be tested with the simple auto generated tests.

One problem with having the "getters" implemented by stored procedures as opposed to user defined functions is that the result of the stored procedure must be stored into a temporary variable before it can be checked. MS SQL Server 2000 has two ways of storing result sets; temporary table and table variable. Result sets of stored procedures can only be stored in temporary tables (tables prefixed with #) and thus those are created and used throughout the tests. Temporary tables are automatically removed by the server when the session terminates and they are also not visible to other users of the database (as long as they are not made global).

Test case 1; INSERT, UPDATE, GET, DELETE.

```
Pseudo code:
* Insert four rows into the table
* Check that the rows were inserted correctly using the "GET" procedure
* Update row 2 with valid values
* Check that the update succeeded
* Update row 3 with duplicate values
* Check that the update failed
* Insert duplicate row
* Check that the insert failed
* Try to delete row4
* Check that the delete succeeded
* Clean up all test data
```

Test case 2, EXC_EXCEPTION, INSERT UPDATE GET DELETE

```
Pseudo code:
* Insert exception A and B
* Insert exception C partly referencing the same entries as exception A
* Check that the rows were inserted correctly
* Update Exception B using strings
* Check that the update succeeded Update Exception B using reference ids
* Check that the update succeeded using the "GET" procedure
* Delete Exception C
* Check that the delete succeeded and only the unused references remain
* Delete Exception A
* Check that all references (now unused) are removed as well
* Clean up all test data
```

The stored procedures For each table there are 4 stored procedures generated; Insert, Update, Get and Delete. The insert and update procedures are responsible for not adding any data which would yield duplicates and therefore they are searching to see if the potential new data already exists before any changes are made. The insert and update procedures return the identity value of the created row, using an output parameter (or if the data already existed, the old identity value is returned).

All procedures except for the Get procedure return their success state as the return value (which if of course not possible in the Get procedure since it returns the requested result set). The Get procedures could be implemented using "User defined functions", which are directly usable in queries (whereas stored procedures must be executed separately), but in manner to be consistent, the functionality was implemented using stored procedures.

The exception model has many fields that are nullable, e.g. exception message, class message, caused by exception and so forth. Thus, the insert and update stored procedures need to be able to handle the case when only a few of the fields should be inserted or updated. In order to be able to update only a few fields of a table without having to write an update stored procedure for every possible combination of fields; the update procedure ignores any null values that it receives. All parameters have null as default and thus the caller of the procedure may omit all parameters that are not to be updated. This complicates the implementation somewhat since the standard states that null are to be treated as undefined. Hence the comparison:

@parameter = null

will never become true – comparisons with null should use the "is" construction, i.e.

@parameter is null.

Since it is impossible to know beforehand if the parameter will be null or contain a value; the update procedure must check for both. This is easiest shown with an example:

```
UPDATE [CIN_CLASSINFORMATION]
    SET [CIN_CLASSNAME]=CASE
        WHEN @CIN_CLASSNAME IS NULL THEN
            [CIN_CLASSNAME]
        ELSE
            @CIN_CLASSNAME
        END
    WHERE [CIN_CLASSID] = @CIN_CLASSID
```

The parameters of the stored procedure is prefixed with an "@".

The final database diagram is found in the results section.

3.7 User interfaces

3.7.1 Methodology

To be able to create good user interfaces, it has to be concluded how the application will be used and by whom. In what environment the application will run is also important and thus the method of how the graphical user interfaces should be created was by discussing it with the team, creating prototypes and further finalize them through feedback.

3.7.2 Analysis

First of all there was to be decided whether the user interface should be a windows application one or a web based one. After a series of meetings it was concluded that the application should have a web interface and be running within the SPIN framework on a WebLogic server. This greatly reduced the possibilities of design choices, since the SPIN project defines how the design should look for all its components to look similar and not cause confusion. Another effect of the SPIN choice is that the internal Amadeus framework Aria (which was previously decided to be used) could not be used, since it relied on a newer version of Java than the SPIN project supported.

The SPIN framework was although a bit complicated to get up and running on the local machine, so while trying to get it all to work a parallel Java Swing view was created just to be able to continue the development of the actual project. The whole purpose of the swing version was to be able to test out the functionality of the application and thus there were absolutely no effort made to make it look good or even be user friendly.

What the prototype however gave, was the confirmation of the general idea of the user interface was acceptable.

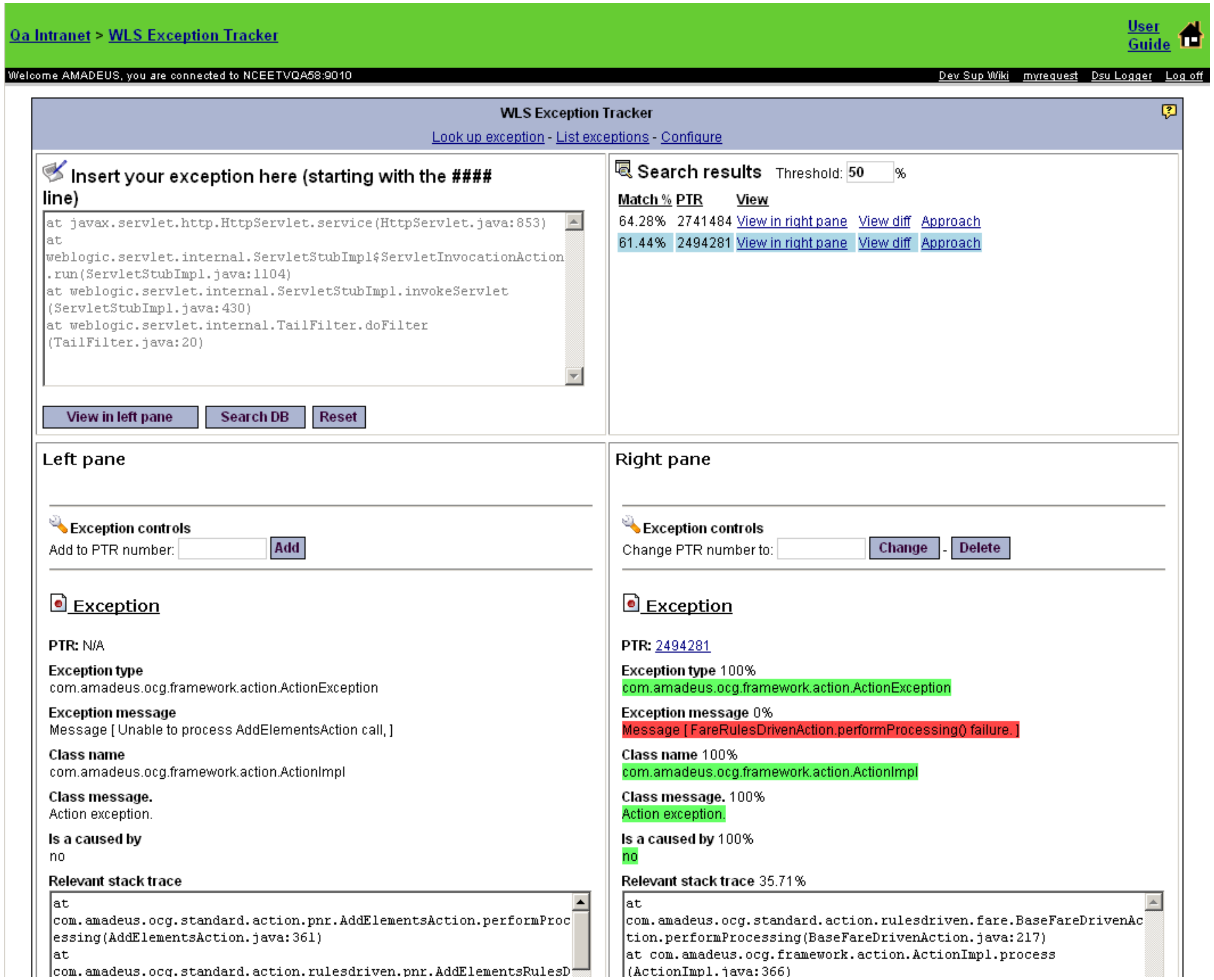
Web interfaces can be made in infinite number of ways using whatever language that feels suitable – it all comes down to how the server is configured. Since it was decided in this case that the project should be a part of SPIN, it was narrowed down to using Java and in particular Java server pages for creating the web pages that made up the user interface. The standard way of making Java based web application is by using Java2 Enterprise Edition (J2EE)[20] and a subset of the methods offered by it was also used in this project. One note to make is that given the small size of this project, no servlets were used in the implementation, but rather the jsp pages used the underlying program classes (the controller layer) directly.

In web interfaces, there is mainly two ways of how user input is handled, through postbacks (which is the old standard way) or through AJAX (a newer and increasingly more popular way). Via post backs, the input will be sent to the server and the whole page will be reloaded and nothing can be done by the user until it is completely recreated. By utilizing AJAX on the other hand, Java script is used to send the input "behind the scenes" and then the response is displayed on the current page directly, also through the use of Java script. The AJAX method is much more complex, but also much more user friendly. What ultimately decided which way to go was how much time that could be afforded to spend on implementing it. The post back method would be quicker, so it was really up to how fast an acceptable AJAX interface could be developed. Fortunately there are open source, free projects that has implemented good frameworks of AJAX and one was decided upon to try out; "JQuery" [21]. The result was above expectation and given that by using Jquery, the project could be completed on time with the superior AJAX interface, the choice was quite obviously to go with AJAX though JQuery. The approach used was to divide the page into different content panes and through AJAX load an internal page which processed the request and displayed the result into the appropriate content pane.

The interface was now constructed by 4 main pages, which internally used 7

[20] http://java.sun.com/javaee/index.jsp
[21] http://jquery.org/

behind the scenes pages that did calculations and other work. See Figure 9 for a screenshot.


Qa Intranet > WLS Exception Tracker
User Guide
Welcome AMADEUS, you are connected to NCEETVQA58:9010
Dev Sup Wiki myrequest Dsu Logger Log off
WLS Exception Tracker
Look up exception - List exceptions - Configure

Insert your exception here (starting with the #### line)

```
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
at
weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction
.run(ServletStubImpl.java:1104)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:430)
at weblogic.servlet.internal.TailFilter.doFilter
(TailFilter.java:20)
```

Search results Threshold: 50 %

Match %	PTR	View		
64.28%	2741484	View in right pane	View diff	Approach
61.44%	2494281	View in right pane	View diff	Approach

Left pane

Exception controls
Add to PTR number: Add

Exception

PTR: N/A

Exception type
com.amadeus.ocg.framework.action.ActionException

Exception message
Message [Unable to process AddElementsAction call,]

Class name
com.amadeus.ocg.framework.action.ActionImpl

Class message.
Action exception.

Is a caused by
no

Relevant stack trace

```
at
com.amadeus.ocg.standard.action.pnr.AddElementsAction.performProc
essing(AddElementsAction.java:361)
at
com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesD
```

Right pane

Exception controls
Change PTR number to: Change - Delete

Exception

PTR: 2494281

Exception type 100%
com.amadeus.ocg.framework.action.ActionException

Exception message 0%
Message [FareRulesDrivenAction.performProcessing() failure.]

Class name 100%
com.amadeus.ocg.framework.action.ActionImpl

Class message. 100%
Action exception.

Is a caused by 100%
no

Relevant stack trace 35.71%

```
at
com.amadeus.ocg.standard.action.rulesdriven.fare.BaseFareDrivenAc
tion.performProcessing(BaseFareDrivenAction.java:217)
at com.amadeus.ocg.framework.action.ActionImpl.process
(ActionImpl.java:366)
```

Figure 9: Screenshot of web user interface

Short description of the pages

- *main.jsp*
  Is the main interface to the program. Here exceptions can be entered and searched for in the internal database. It has 6 <div> panels on it in which it loads the result of the users actions through AJAX.

  - *displayException.jsp*
    Renders a view of an exception in terms of its features. If another exception is loaded, the similarity is also displayed using percentage and color. This internal page is also used in the listExceptions.jsp page.
  - *displayDiff.jsp*
    Displays the difference between two exceptions using a traditional Diff view, using colors.
  - *changePTRAssociation.jsp*
    Executes the functions of changing the PTR association of the loaded exception in the internal database .
  - *deleteException.jsp*
    Deletes the loaded exception from the internal database.
  - *addToDatabase.jsp*
    Adds the entered exception to the internal database.
  - *serachException.jsp*
    Performs the search of similar exceptions to the one entered and displays matches in a sorted list along with match percentages.

- *listExceptions.jsp*
  Is a search page for managing the exceptions stored in the internal database.

  - *listExceptionResult.jsp*
    Performs the search and displays the result in a list.

- *errorPage.jsp*
  Is displayed if the application encounters an unexpected error in any page.

- *configure.jsp*
  Configures the regular expressions used to extract the exception features.

For screenshots, see the results chapter.

4 Results

The result of the master thesis is the best found methodology of matching Java exceptions, employed within a web based application. Each sub part of the methodology and path to put it all together is presented in different sub chapters within this result chapter.

4.1 Exception definition

An exception is split up in a number of exception features, a stack trace and zero or more caused by exceptions, which in turn have their own features, stack trace and zero or more caused by exceptions.

Features

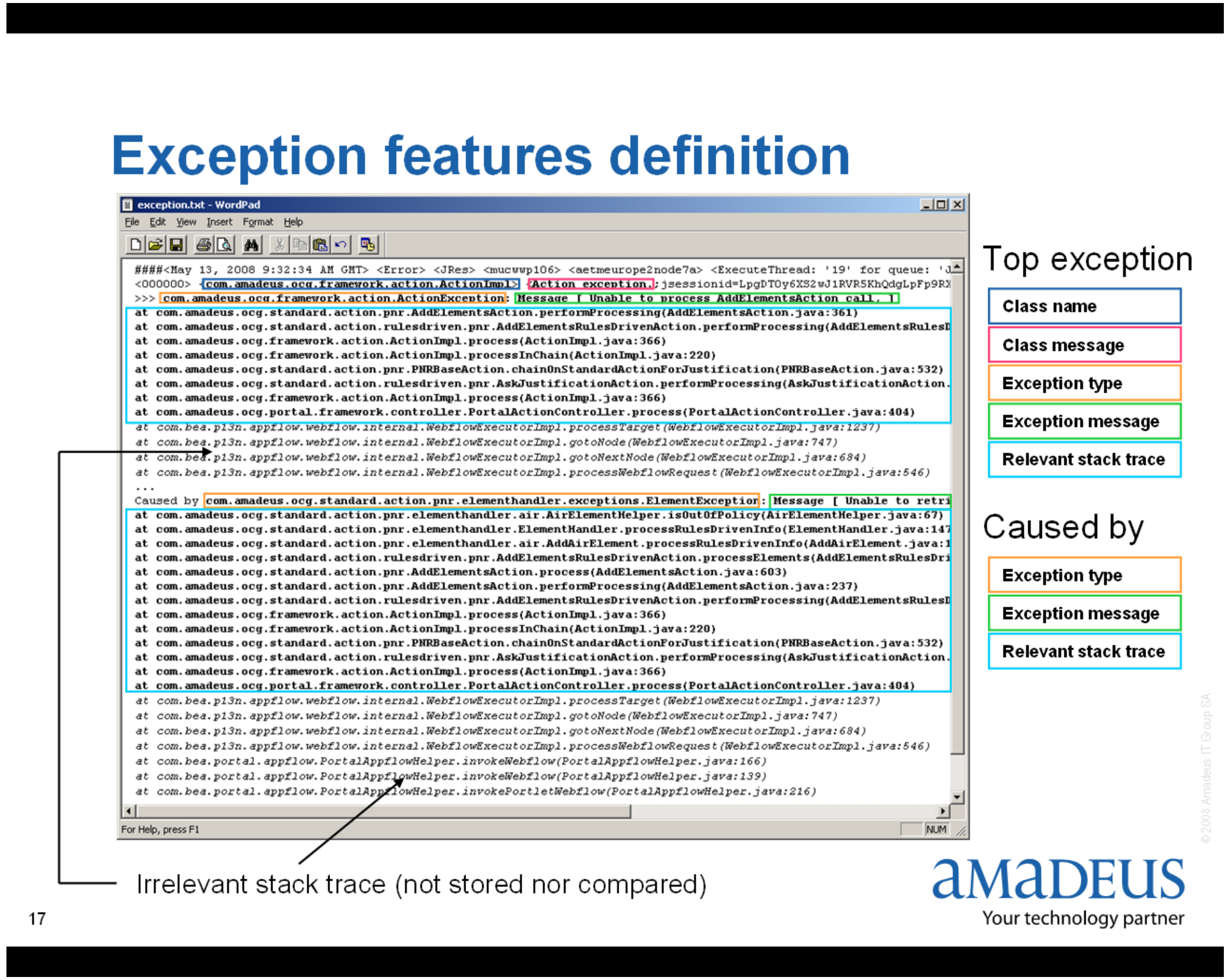
Class name: The class which threw the exception

Class message: A message associated with the class and exception

Exception type: The exception type

Exception message: The message associated with the exception type and instance

Relevant stack trace: The stack trace that is relevant for solving the problem; e.g. the stack that is from the application itself and not from the containing server.



Exception features definition

```
####<May 13, 2008 9:32:34 AM GMT> <Error> <JRes> <mucwwp106> <aetmeurope2node7a> <ExecuteThread: '19' for queue: 'J
<000000> <com.amadeus.ocg.framework.action.ActionImpl> <Action exception.;jsessionid=LpgDTOy6XS2wJ1RVR5KhQdgLpFp9RX
>>> com.amadeus.ocg.framework.action.ActionException: Message [ Unable to process AddElementsAction call. ]
at com.amadeus.ocg.standard.action.pnr.AddElementsAction.performProcessing(AddElementsAction.java:361)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesDrivenAction.performProcessing(AddElementsRulesD
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.framework.action.ActionImpl.processInChain(ActionImpl.java:220)
at com.amadeus.ocg.standard.action.pnr.PNRBaseAction.chainOnStandardActionForJustification(PNRBaseAction.java:532)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AskJustificationAction.performProcessing(AskJustificationAction.
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.portal.framework.controller.PortalActionController.process(PortalActionController.java:404)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processTarget(WebflowExecutorImpl.java:1237)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNode(WebflowExecutorImpl.java:747)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNextNode(WebflowExecutorImpl.java:684)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processWebflowRequest(WebflowExecutorImpl.java:546)
...
Caused by com.amadeus.ocg.standard.action.pnr.elementhandler.exceptions.ElementException: Message [ Unable to retri
at com.amadeus.ocg.standard.action.pnr.elementhandler.air.AirElementHelper.isOutOfPolicy(AirElementHelper.java:67)
at com.amadeus.ocg.standard.action.pnr.elementhandler.ElementHandler.processRulesDrivenInfo(ElementHandler.java:147
at com.amadeus.ocg.standard.action.pnr.elementhandler.air.AddAirElement.processRulesDrivenInfo(AddAirElement.java:1
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesDrivenAction.processElements(AddElementsRulesDri
at com.amadeus.ocg.standard.action.pnr.AddElementsAction.process(AddElementsAction.java:603)
at com.amadeus.ocg.standard.action.pnr.AddElementsAction.performProcessing(AddElementsAction.java:237)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesDrivenAction.performProcessing(AddElementsRulesD
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.framework.action.ActionImpl.processInChain(ActionImpl.java:220)
at com.amadeus.ocg.standard.action.pnr.PNRBaseAction.chainOnStandardActionForJustification(PNRBaseAction.java:532)
at com.amadeus.ocg.standard.action.rulesdriven.pnr.AskJustificationAction.performProcessing(AskJustificationAction.
at com.amadeus.ocg.framework.action.ActionImpl.process(ActionImpl.java:366)
at com.amadeus.ocg.portal.framework.controller.PortalActionController.process(PortalActionController.java:404)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processTarget(WebflowExecutorImpl.java:1237)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNode(WebflowExecutorImpl.java:747)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.gotoNextNode(WebflowExecutorImpl.java:684)
at com.bea.p13n.appflow.webflow.internal.WebflowExecutorImpl.processWebflowRequest(WebflowExecutorImpl.java:546)
at com.bea.portal.appflow.PortalAppflowHelper.invokeWebflow(PortalAppflowHelper.java:166)
at com.bea.portal.appflow.PortalAppflowHelper.invokeWebflow(PortalAppflowHelper.java:139)
at com.bea.portal.appflow.PortalAppflowHelper.invokePortletWebflow(PortalAppflowHelper.java:216)
```

Figure 10: Exception features

4.2 Matching rules

When comparing the exceptions, a set of rules on how to compare the different exception features is needed. The resulting rule chosen is to compare the top exception in the chain of caused by exceptions and the last one at the bottom of the chain. The features are compared by filtering out all exceptions that does not have the exact exception type and then applying fuzzy string matching on the rest of the features. The result is the average match metric, which is normalized to be represented in a match percentage.

There is probably room for improvement in the matching rules if a more optimal solution is needed. However, this rule was good enough for meeting the performance criteria of the application implemented to perform the matching job.

4.3 Fuzzy string matching algorithm

Two algorithms are possible to use in the application: the thresholded lazy evaluating Levenshtein algorithm and the thresholded small memory Levenshtein algorithm. Both calculates the standard Levenshtein edit distance between the strings, i.e. how many inserts, substitutions and deletions of characters that are needed to convert the first string into the other. The thresholding results in the calculations being aborted if the strings are detected to be too dissimilar. The strings are then considered infinite unequal. Since unsimilar exceptions are irrelevant for the application, this greatly speeds up the process without lowering the quality of the results.

The lazy evaluation speeds up the process by only calculating the necessary values to reach the result, however it might run out of memory in the process and thus the " Small memory Levenshtein algorithm" is possible to use instead.

4.4 Program structure

The application that performs the matching job is structured as a Model View Controller application, with the modifications that the view is completely detached from the model (necessary to make a web interface) and an extra utility layer is added, which all other layers can access. The model, utility and controller namespaces are made up of ordinary Java classes and the presentation layer is made up of Java server script pages, see Figures 11,12,13 and 14 for details.

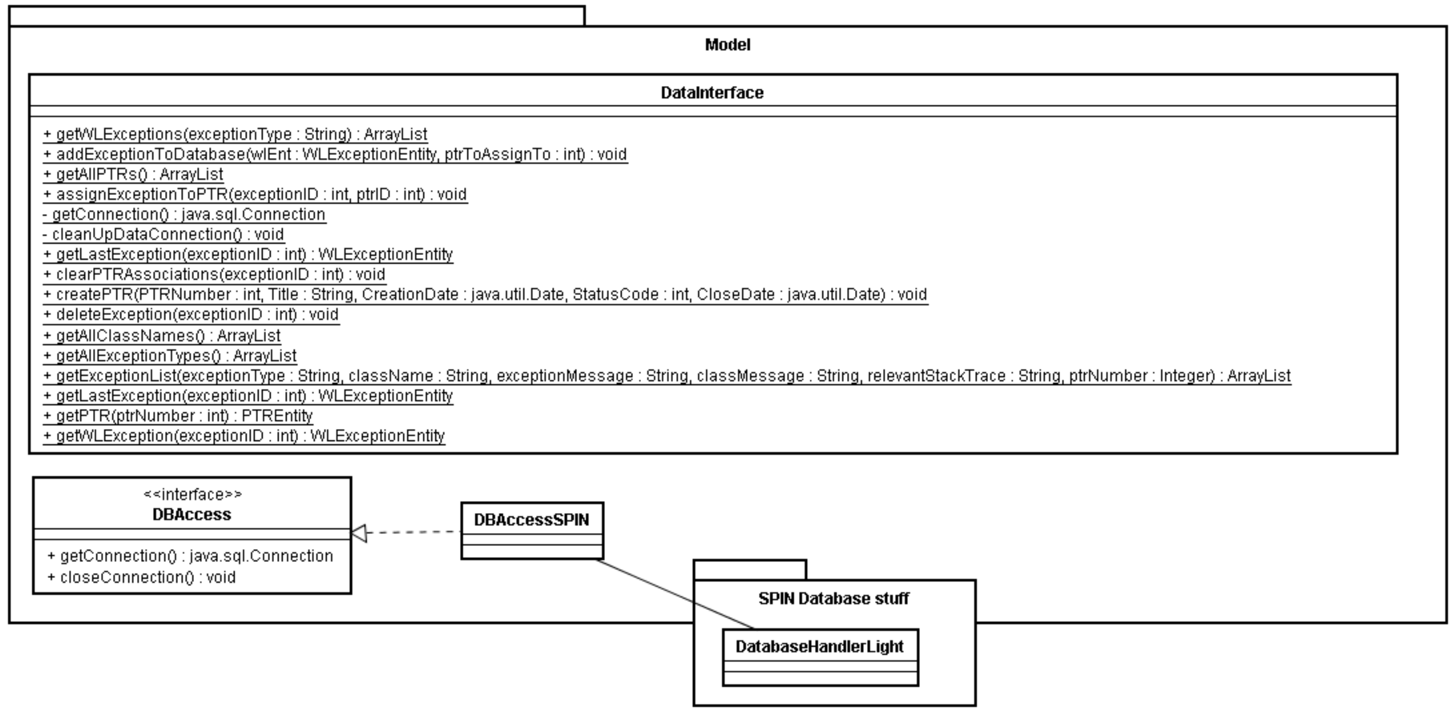

Model
DataInterface
+ getWLExceptions(exceptionType : String) : ArrayList
+ addExceptionToDatabase(wlEnt : WLExceptionEntity, ptrToAssignTo : int) : void
+ getAllPTRs() : ArrayList
+ assignExceptionToPTR(exceptionID : int, ptrID : int) : void
- getConnection() : java.sql.Connection
- cleanUpDataConnection() : void
+ getLastException(exceptionID : int) : WLExceptionEntity
+ clearPTRAssociations(exceptionID : int) : void
+ createPTR(PTRNumber : int, Title : String, CreationDate : java.util.Date, StatusCode : int, CloseDate : java.util.Date) : void
+ deleteException(exceptionID : int) : void
+ getAllClassNames() : ArrayList
+ getAllExceptionTypes() : ArrayList
+ getExceptionList(exceptionType : String, className : String, exceptionMessage : String, classMessage : String, relevantStackTrace : String, ptrNumber : Integer) : ArrayList
+ getLastException(exceptionID : int) : WLExceptionEntity
+ getPTR(ptrNumber : int) : PTREntity
+ getWLException(exceptionID : int) : WLExceptionEntity
<<interface>>
DBAccess
+ getConnection() : java.sql.Connection
+ closeConnection() : void
DBAccessSPIN
SPIN Database stuff
DatabaseHandlerLight

Figure 11: UML class diagram of the Data model

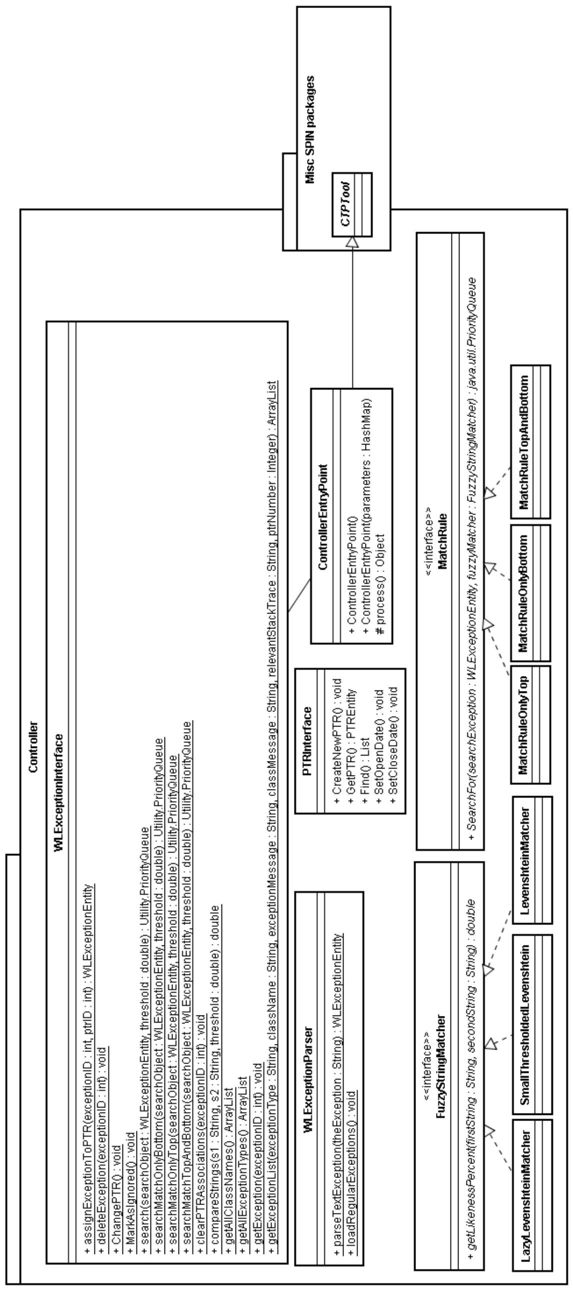

Controller
WLExceptionInterface
+ assignExceptionToPTR(exceptionID : int, ptrID : int) : WLExceptionEntity
+ deleteException(exceptionID : int) : void
+ ChangePTR() : void
+ MarkAsIgnored() : void
+ search(searchObject : WLExceptionEntity, threshold : double) : Utility.PriorityQueue
+ searchMatchOnlyBottom(searchObject : WLExceptionEntity, threshold : double) : Utility.PriorityQueue
+ searchMatchOnlyTop(searchObject : WLExceptionEntity, threshold : double) : Utility.PriorityQueue
+ searchMatchTopAndBottom(searchObject : WLExceptionEntity, threshold : double) : Utility.PriorityQueue
+ clearPTRAssociations(exceptionID : int) : void
+ compareStrings(s1 : String, s2 : String, threshold : double) : double
+ getAllClassNames() : ArrayList
+ getAllExceptionTypes() : ArrayList
+ getException(exceptionID : int) : void
+ getExceptionList(exceptionType : String, className : String, exceptionMessage : String, classMessage : String, relevantStackTrace : String, ptrNumber : Integer) : ArrayList
WLExceptionParser
+ parseTextException(theException : String) : WLExceptionEntity
+ loadRegularExceptions() : void
PTRInterface
+ CreateNewPTR() : void
+ GetPTR() : PTREntity
+ Find() : List
+ SetOpenDate() : void
+ SetCloseDate() : void
ControllerEntryPoint
+ ControllerEntryPoint()
+ ControllerEntryPoint(parameters : HashMap)
# process() : Object
Misc SPIN packages
CTPTool
<<interface>>
FuzzyStringMatcher
+ getLikenessPercent(firstString : String, secondString : String) : double
LazyLevenshteinMatcher
SmallThresholdedLevenshtein
LevenshteinMatcher
<<interface>>
MatchRule
+ SearchFor(searchException : WLExceptionEntity, fuzzyMatcher : FuzzyStringMatcher) : java.util.PriorityQueue
MatchRuleOnlyTop
MatchRuleOnlyBottom
MatchRuleTopAndBottom

Figure 12: UML class diagram of the Controller layer

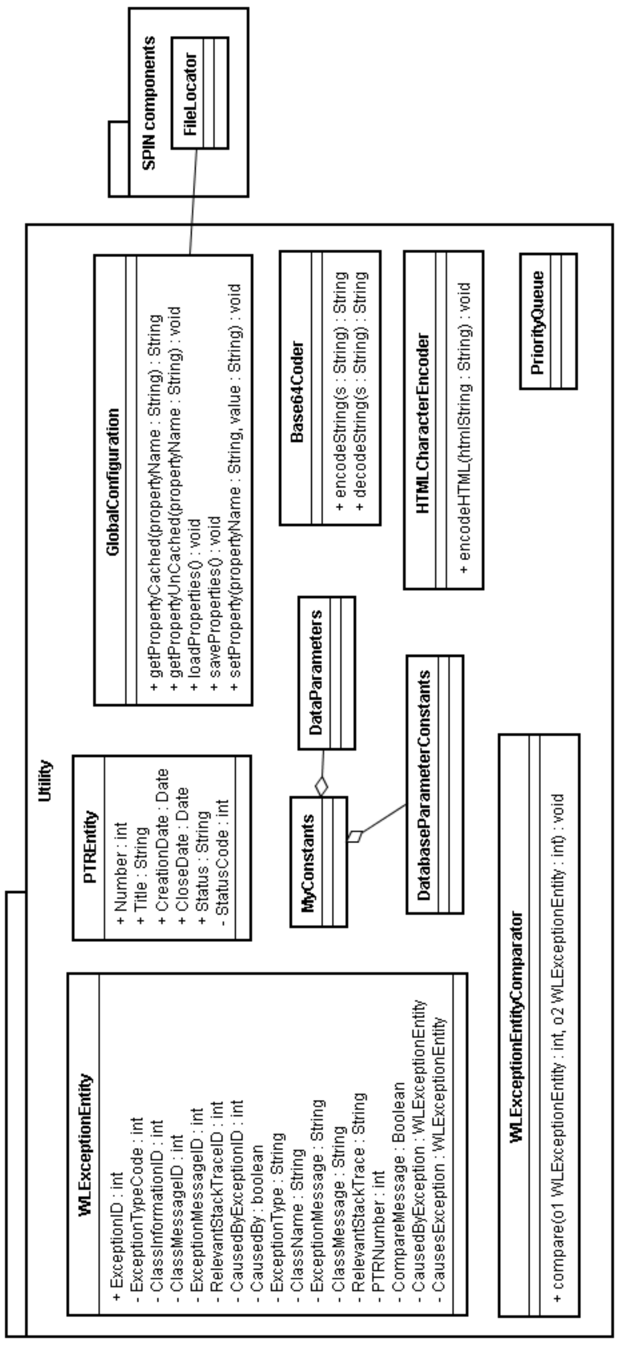

SPIN components
FileLocator
Utility
GlobalConfiguration
+ getPropertyCached(propertyName : String) : String
+ getPropertyUnCached(propertyName : String) : void
+ loadProperties() : void
+ saveProperties() : void
+ setProperty(propertyName : String, value : String) : void
Base64Coder
+ encodeString(s : String) : String
+ decodeString(s : String) : String
HTMLCharacterEncoder
+ encodeHTML(htmlString : String) : void
PriorityQueue
DataParameters
DatabaseParameterConstants
MyConstants
PTREntity
+ Number : int
+ Title : String
+ CreationDate : Date
+ CloseDate : Date
+ Status : String
- StatusCode : int
WLExceptionEntity
+ ExceptionID : int
- ExceptionTypeCode : int
- ClassInformationID : int
- ClassMessageID : int
- ExceptionMessageID : int
- RelevantStackTraceID : int
- CausedByExceptionID : int
- CausedBy : boolean
- ExceptionType : String
- ClassName : String
- ExceptionMessage : String
- ClassMessage : String
- RelevantStackTrace : String
- PTRNumber : int
- CompareMessage : Boolean
- CausedByException : WLExceptionEntity
- CausesException : WLExceptionEntity
WLExceptionEntityComparator
+ compare(o1 WLExceptionEntity : int, o2 WLExceptionEntity : int) : void

Figure 13: UML class diagram of the Utility layer

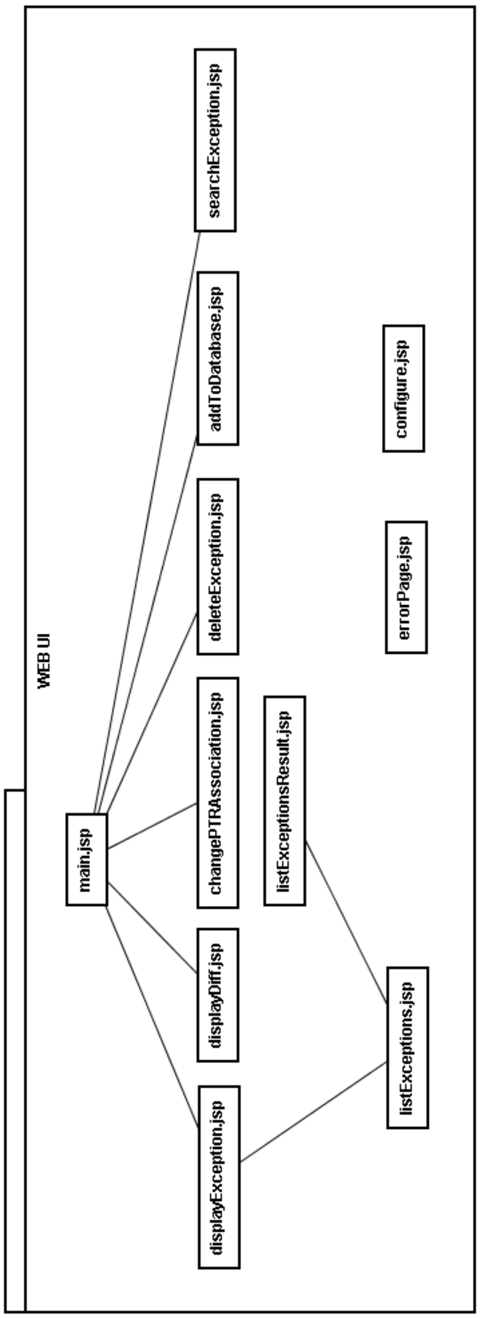
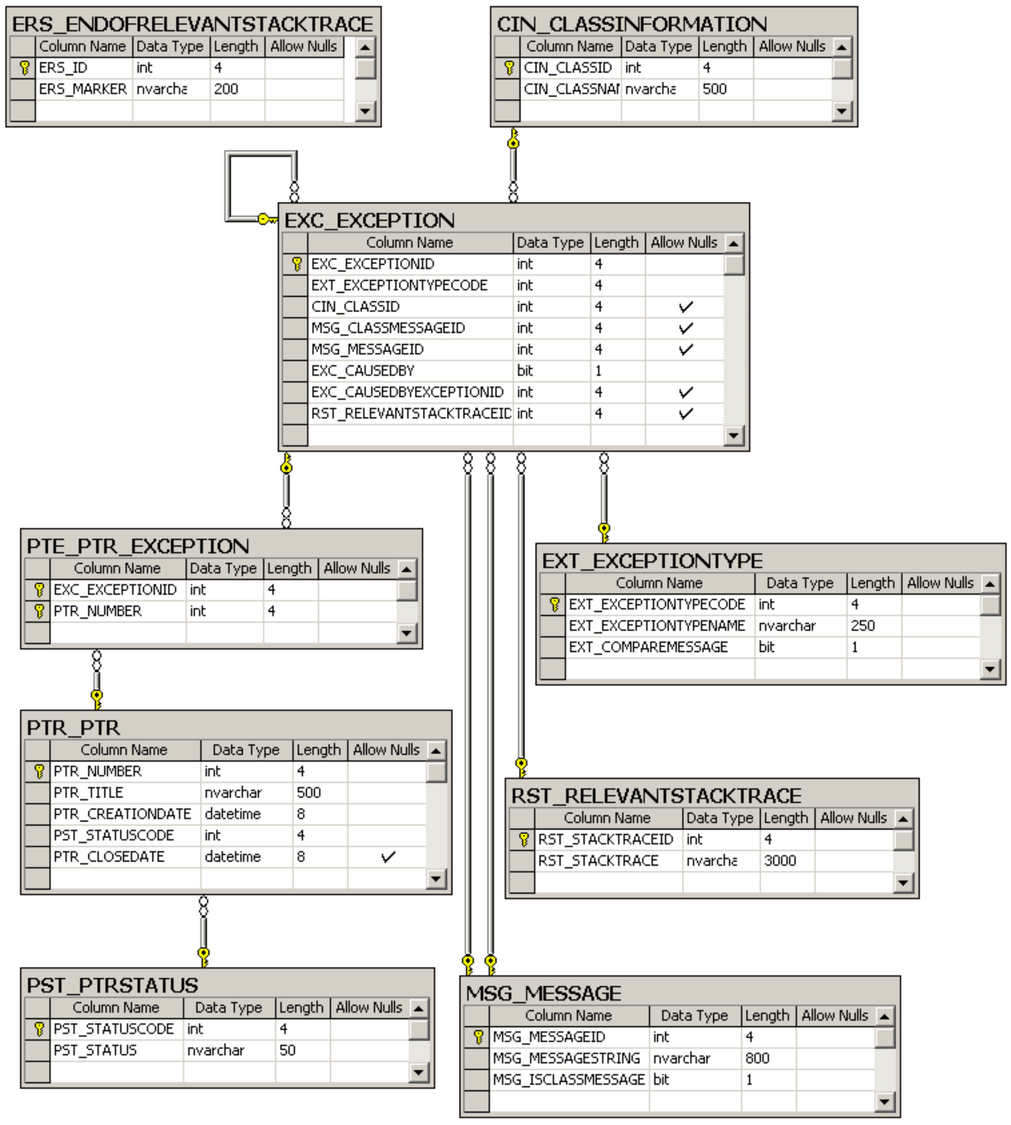


Figure 14: Diagram of the Presentation layer

4.5 Storage (database) model

The storage model is implemented in a MS SQL Server 2000 and thus uses T-SQL. Stored procedures are implemented for all data modifications. See Figure 15



ERS_ENDOFRELEVANTSTACKTRACE

Column Name	Data Type	Length	Allow Nulls
ERS_ID	int	4	
ERS_MARKER	nvarcha	200	

CIN_CLASSINFORMATION

Column Name	Data Type	Length	Allow Nulls
CIN_CLASSID	int	4	
CIN_CLASSNAI	nvarcha	500	

EXC_EXCEPTION

Column Name	Data Type	Length	Allow Nulls
EXC_EXCEPTIONID	int	4	
EXT_EXCEPTIONTYPECODE	int	4	
CIN_CLASSID	int	4	✓
MSG_CLASSMESSAGEID	int	4	✓
MSG_MESSAGEID	int	4	✓
EXC_CAUSEDBY	bit	1	
EXC_CAUSEDBYEXCEPTIONID	int	4	✓
RST_RELEVANTSTACKTRACEIC	int	4	✓

PTE_PTR_EXCEPTION

Column Name	Data Type	Length	Allow Nulls
EXC_EXCEPTIONID	int	4	
PTR_NUMBER	int	4	

EXT_EXCEPTIONTYPE

Column Name	Data Type	Length	Allow Nulls
EXT_EXCEPTIONTYPECODE	int	4	
EXT_EXCEPTIONTYPENAME	nvarchar	250	
EXT_COMPAREMESSAGE	bit	1	

PTR_PTR

Column Name	Data Type	Length	Allow Nulls
PTR_NUMBER	int	4	
PTR_TITLE	nvarchar	500	
PTR_CREATIONDATE	datetime	8	
PST_STATUSCODE	int	4	
PTR_CLOSEDATE	datetime	8	✓

RST_RELEVANTSTACKTRACE

Column Name	Data Type	Length	Allow Nulls
RST_STACKTRACEID	int	4	
RST_STACKTRACE	nvarcha	3000	

PST_PTRSTATUS

Column Name	Data Type	Length	Allow Nulls
PST_STATUSCODE	int	4	
PST_STATUS	nvarchar	50	

MSG_MESSAGE

Column Name	Data Type	Length	Allow Nulls
MSG_MESSAGEID	int	4	
MSG_MESSAGESTRING	nvarchar	800	
MSG_ISCLASSMESSAGE	bit	1	

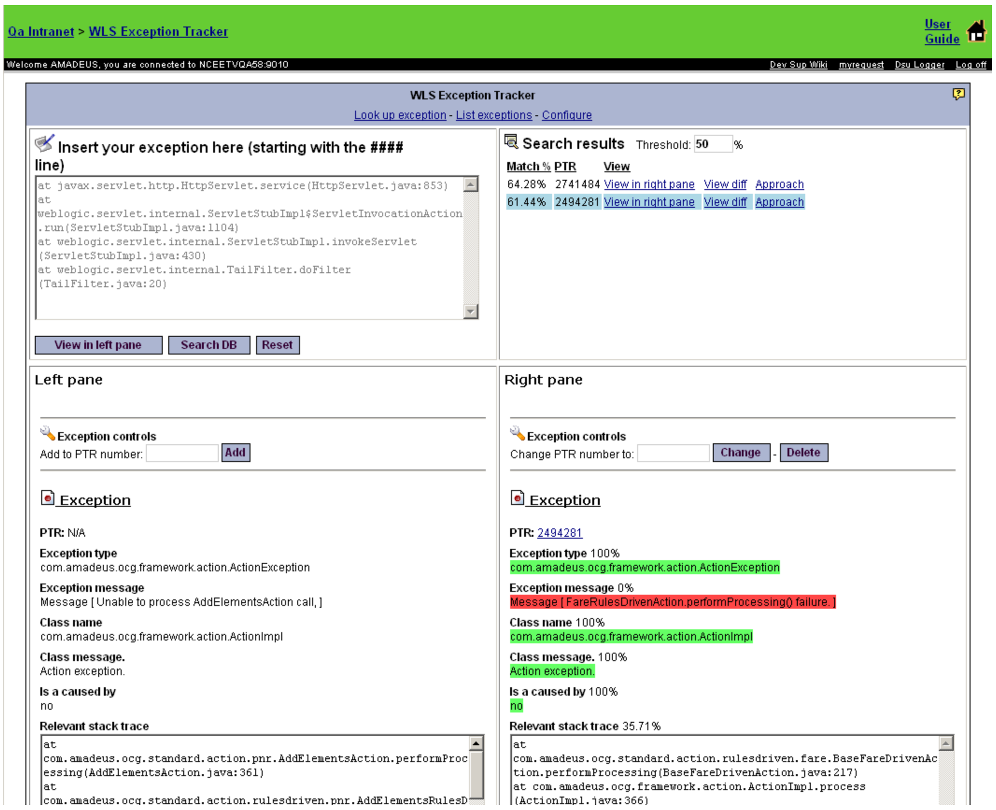
Figure 15: Database diagram

4.6 Application user interface

4.6.1 Web application

The user interface to the application created to do the matching job, is web based and written in Java 1.4 and Java server pages. The layout conforms to the SPIN policy. It utilizes jQuery[22] to implement AJAX support. See screenshots in Figure 16, 17 and 18.

[22]http://www.jquery.org



Qa Intranet > WLS Exception Tracker

User Guide

Welcome AMADEUS, you are connected to NCEETVQA58:9010

Dev Sup Wiki myrequest Dsu Logger Log off

WLS Exception Tracker

Look up exception - List exceptions - Configure

Insert your exception here (starting with the #### line)

```
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
at
weblogic.servlet.internal.ServletStubImpl$ServletInvocationAction
.run(ServletStubImpl.java:1104)
at weblogic.servlet.internal.ServletStubImpl.invokeServlet
(ServletStubImpl.java:430)
at weblogic.servlet.internal.TailFilter.doFilter
(TailFilter.java:20)
```

View in left pane | Search DB | Reset

Search results Threshold: 50 %

Match %	PTR	View		
64.28%	2741484	View in right pane	View diff	Approach
61.44%	2494281	View in right pane	View diff	Approach

Left pane

Exception controls
Add to PTR number: Add

Exception

PTR: N/A

Exception type
com.amadeus.ocg.framework.action.ActionException

Exception message
Message [Unable to process AddElementsAction call,]

Class name
com.amadeus.ocg.framework.action.ActionImpl

Class message.
Action exception.

Is a caused by
no

Relevant stack trace

```
at
com.amadeus.ocg.standard.action.pnr.AddElementsAction.performProc
essing(AddElementsAction.java:361)
at
com.amadeus.ocg.standard.action.rulesdriven.pnr.AddElementsRulesD
```

Right pane

Exception controls
Change PTR number to: Change - Delete

Exception

PTR: 2494281

Exception type 100%
com.amadeus.ocg.framework.action.ActionException

Exception message 0%
Message [FareRulesDrivenAction.performProcessing() failure.]

Class name 100%
com.amadeus.ocg.framework.action.ActionImpl

Class message. 100%
Action exception.

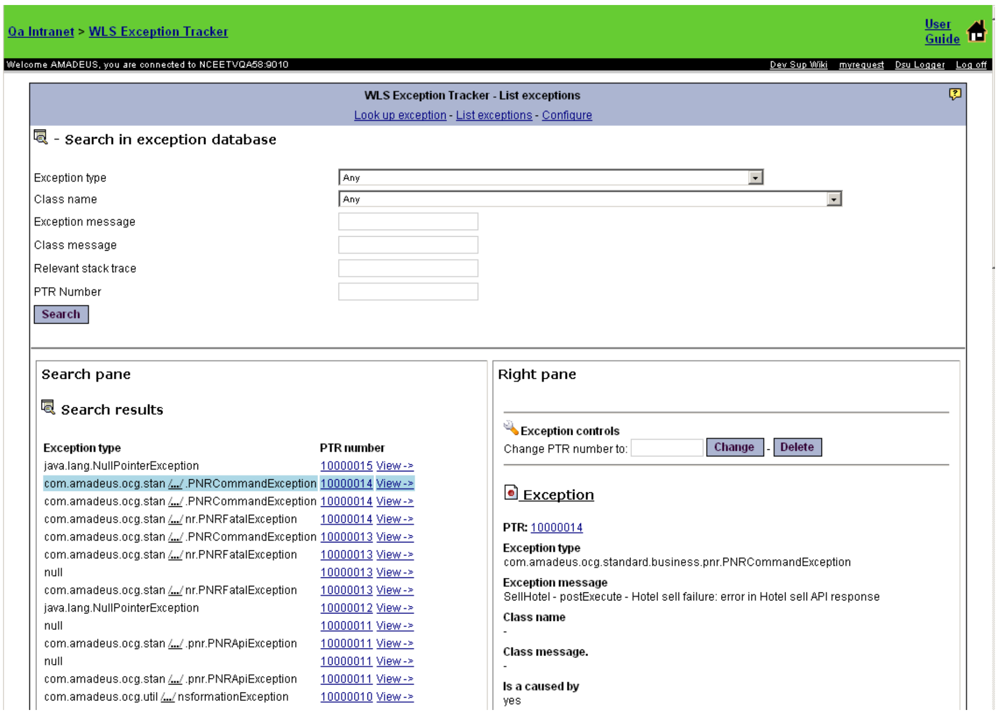
Is a caused by 100%
no

Relevant stack trace 35.71%

```
at
com.amadeus.ocg.standard.action.rulesdriven.fare.BaseFareDrivenAc
tion.performProcessing(BaseFareDrivenAction.java:217)
at com.amadeus.ocg.framework.action.ActionImpl.process
(ActionImpl.java:366)
```

Here, an exception has been entered, viewed and searched for. A possible candidate has been selected for viewing the differences.

Figure 16: Main page



Qa Intranet > WLS Exception Tracker

User Guide

Welcome AMADEUS, you are connected to NCEETVQA58:9010 — Dev Sup Wiki — myrequest — Dsu Logger — Log off

WLS Exception Tracker - List exceptions

Look up exception - List exceptions - Configure

- Search in exception database

Exception type	Any
Class name	Any
Exception message	
Class message	
Relevant stack trace	
PTR Number	

Search

Search pane

Search results

Exception type	PTR number	
java.lang.NullPointerException	10000015	View ->
com.amadeus.ocg.stan /.../ .PNRCommandException	10000014	View ->
com.amadeus.ocg.stan /.../ .PNRCommandException	10000014	View ->
com.amadeus.ocg.stan /.../ nr.PNRFatalException	10000014	View ->
com.amadeus.ocg.stan /.../ .PNRCommandException	10000013	View ->
com.amadeus.ocg.stan /.../ nr.PNRFatalException	10000013	View ->
null	10000013	View ->
com.amadeus.ocg.stan /.../ nr.PNRFatalException	10000013	View ->
java.lang.NullPointerException	10000012	View ->
null	10000011	View ->
com.amadeus.ocg.stan /.../ .pnr.PNRApiException	10000011	View ->
null	10000011	View ->
com.amadeus.ocg.stan /.../ .pnr.PNRApiException	10000011	View ->
com.amadeus.ocg.util /.../ nsformationException	10000010	View ->

Right pane

Exception controls

Change PTR number to: Change - Delete

Exception

PTR: 10000014

Exception type
com.amadeus.ocg.standard.business.pnr.PNRCommandException

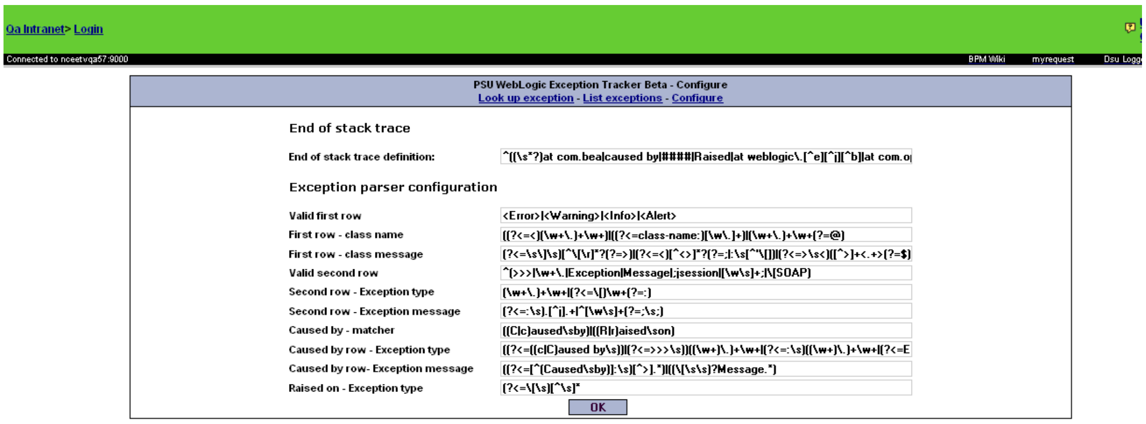
Exception message
SellHotel - postExecute - Hotel sell failure: error in Hotel sell API response

Class name
-

Class message.
-

Is a caused by
yes

Figure 17: List of exceptions in database



Qa Intranet> Login

Connected to nceetvqa57:9000 — BPM Wiki — myrequest — Dsu Logg

PSU WebLogic Exception Tracker Beta - Configure

Look up exception - List exceptions - Configure

End of stack trace

```
End of stack trace definition:   ^((\s*?)at com.bea|caused by|####|Raised|at weblogic\.[^e][^j][^b]|at com.o
```

Exception parser configuration

```
Valid first row                       <Error>|<Warning>|<Info>|<Alert>
First row - class name                ((?<=<)(\w+\.)+\w+)|((?<=class-name:)[\w\.]+)|(\w+\.)+\w+(?=@)
First row - class message             (?<=\s\]\s)[^\[\r]*?(?=>)|(?<=<)[^<>]*?(?=;|:\s[^"\[])|(?<=>\s<)([^>]+<.+>(?=$)
Valid second row                      ^(>>>|\w+\.|Exception|Message|;jsession|[\w\s]+;|\[SOAP)
Second row - Exception type           (\w+\.)+\w+|(?<=\[)\w+(?=:)
Second row - Exception message        (?<=:\s).[^j].+|^[\w\s]+(?=;\s;)
Caused by - matcher                   ((C|c)aused\sby)|((R|r)aised\son)
Caused by row - Exception type        ((?<=((c|C)aused by\s))|(?<=>>>\s))((\w+)\.)+\w+|(?<=:\s)((\w+)\.)+\w+|(?<=E
Caused by row- Exception message      ((?<=[^(Caused\sby)]:\s)[^>].*)|((\[\s\s)?Message.*)
Raised on - Exception type            (?<=\[\s)[^\s]*
```

OK

Figure 18: Configuration of exception features

4.6.2 Web service

The web service was unfortunately not implemented due to lack of time.

5 Discussion

The application fulfills the intended requirements and is now used on a daily basis. As a performance goal, the matching process is fast enough, but it is yet unknown how it will perform when the database gets larger. Even more studies can easily be done on how to optimize the matching rules and algorithms further to improve the speed. How the system will perform in cooperation with the SWAT system is not investigated and it might need some adaptation for it to work in reasonable performance. Unfortunately there was no time to implement the web service. However, given the modular MVC structure used in the program it should not be difficult at all to implement one.

If the syntax of handled exceptions are even further expanded, the method used with precompiled regular expressions can be unfeasible to maintain[23], the system might need to be rewritten to parse exceptions in two steps: first determine what type of exception syntax it is, and then apply the appropriate regular expressions for it.

6 Conclusion

The method of fuzzy string matching works good in the area of exception matching, using the proposed definition of exception features. The performance needed seems to be fully met in the implementation. However, how it scales as the database grows bigger is yet unknown.

Suggestions of further development is to implement the link to SWAT, implement the web service interface, and also to look at the possibility of using a weighted sum instead of an average in the matching rule to get even smarter matching results. And finally, to update the parsing regular expressions to cover all new exception syntaxes encountered.

[23]The regular expressions would at some point become too complicated for manual management

7 Bibliography

L. Allison, *Lazy Dynamic-Programming can be Eager*, Information processing letters 43(4) 207-212,
http://www.csse.monash.edu.au/~lloyd/tildeStrings/Alignment/92.IPL.ps,
2008-07-01

Top coder software, *String Distance1.0 Component Specification,*
http://software.topcoder.com/catalog/document?id=8457494,
2008-07-01

Wikipedia, *Boyer-Moore algorithm,*
http://en.wikipedia.org/w/index.php?title=Boyer%E2%80%93Moore_string_search_algorithm&oldid=221497526,
2008-07-01

Wikipedia, *Damerau-Levenshtein algorithm,*
http://en.wikipedia.org/w/index.php?title=Damerau%E2%80%93Levenshtein_distance&oldid=218621877,
2008-07-01

Wikipedia, *Jaro-Winkler algorithm,*
http://en.wikipedia.org/w/index.php?title=Jaro-Winkler_distance&oldid=216235867,
2008-07-01

Wikipedia, *Kohonen Map,*
http://en.wikipedia.org/w/index.php?title=Self-organizing_map&oldid=206987930,
2008-07-01

Wikipedia, *Levenshtein distance,*
http://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=222332508,
2008-07-01

Wikipedia, *Needleman-Wunsch algorithm,*
http://en.wikipedia.org/w/index.php?title=Needleman%E2%80%93Wunsch_algorithm&oldid=215012856,
2008-07-01

Wikipedia, *Regular expressions,*
http://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=347265226,
2008-07-01

8 Literature

Stephen Stelting, *robust JAVA Exception Handling*, Testing and Debugging, Prentice Hall, 2005

9 Glossary and index

AJAX: Technology for dealing with user input on homepages without using post backs

ARIA: Internal Amadeus framework for web 2.0 interfaces

"Caused by" exception: An exception that causes another exception to occur.

DBMS: Database management system

GUI: Graphical user interface

Matching rule: A set of definitions on how the features of an exception should be combined to determine similarities and search matches.

NRE: Non regression testing team

PSU: Production support team

PTR: Problem tracking record

Regular expression: A notion for matching text patterns

Relevant stack trace: The part of the stack trace that is from the application code and not referring to the platform (WebLogic)

SPIN: Internal Amadeus software library and policy collection for development

SWAT: Internal Amadeus application for handling exception logs etc

WebLogic: Platform for running J2EE applications

Index