Development of a Tool for the Evaluation of Change Requests

Diploma Thesis

Author:	Lars Grammel
Date submitted:	February 22, 2007
First Reviewer:	Prof. Dr. rer. nat. Horst Lichter
Second Reviewer:	Prof. Dr.-Ing. Stefan Kowalewski
Supervisor:	Dipl.-Inform. Holger Schackmann
Created at:	Research Group Software Construction Prof. Dr. rer. nat. Horst Lichter Faculty of Mathematics, Computer Sciences and Natural Sciences RWTH Aachen University

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Aachen, 22. Februar 2007

Contents

Chapter 1

Introduction

The main goal of software engineering is the efficient development and maintenance of high quality software systems. In order to achieve this, software systems as well as software processes must be evaluated methodically. The results of these evaluations can then be used for controlling, managing, planing and improving the development of software systems.

An appropriate data entry as well as measurements that are based on it provide a well-founded basis for such evaluations. In the case of software systems, a lot of data is automatically available in the form of source code. Other data like the number of errors must be collected separately. In the case of software processes and used resources the data must be acquired separately, too.

The benefits that result from the measurements should exceed the costs of the measurements and the associated data entry. For reducing these costs, the data should preferably be entered directly into an electronic system and the measurements should be automated to a large extent. For measurements that are based on the source code of software systems, this is already the case.

Another artifact that is often already available electronically are change requests. A measurement of change requests can provide useful information about the software systems as well as the software processes and the resource usage.

Tools for the administration of change requests often provide some evaluations of change requests, too. But these evaluations are rather limited. Thus, an automatic evaluation of change requests is only possible to a limited extend so far.

Task Description

The popular open-source tool Bugzilla is used at Kisters AG for the administration of change requests. The tool has been complemented by several extensions, among other things for an improved dependency tracking between different development projects and for an extended state administration.

In the context of process improvement measures first requirements for metric evaluations over the change requests that are administered in Bugzilla have been formulated. These requirements should be completed and analyzed at first. Thereby the characteristics of the different product developments and the embedding into the complete development process have to be considered.

On this basis requirements for tool support for metric calculation and for the generation of charts should be determined. A solution for tool support should be designed and implemented. For the design of the tool it should be considered that the tool should be extensible by additional metrics and charts and that the effort for administration and maintenance of the developed solution should be low.

The diploma thesis is therefore structured into the following tasks:

1. Familiarization with the topic on the basis of the literature [SHT05, Buh04, LW00, BUGb]

2. Elicitation and analysis of the requirement for an evaluation tool

3. Analysis of the existing evaluation possibilities provided by Bugzilla

4. Design and implementation of a tool for the evaluation of change requests

5. Evaluation of the developed solution

6. In parallel to the tasks 1-5, the results should be documented in written form

Cooperation partner for this diploma thesis is Kisters AG, Aachen. Kisters AG is a growing, internationally expanding company that provides solutions for the areas resource management systems, environment health safety, information technology and environmental consulting. The software solutions are organized in a growing product portfolio of related products.

Structure of the Thesis

In chapter two, the basic terms that are used throughout this thesis are defined. The next chapter provides an evaluation of the existing tools for calculating metrics over change requests. These two chapters contain the foundations for the requirements of the tool that has been developed. These are presented in chapter four. The software process that was used for the development of the tool is outlined in chapter five. Chapter six contains information about the architecture and the design of the tool. In chapter seven, the software process, the developed tool and the used metrics are evaluated. The last chapter finally provides a summary of this thesis and a perspective on future extensions.

Chapter 2

Basic Terms

In this chapter, important basic terms from the areas of software engineering, software metrics and charts that are used in the following chapters are explained and defined.

A **software project** is temporary activity that is characterized by having a start date, the objective to develop a software system, constraints, established responsibilities, a budget and schedule, and a completion date [Tha97]. The objective and the constraints of a software project may be unstable [LL06, p. 90]. After the software solution has been delivered, the software project is finished.

A **software product** is a software system that is sold to one or multiple customers and that is delivered in several releases. According to Sneed [SHT05], the different releases are developed in multiple software projects, with each new release based on the previous one.

The general process of changing a software system after delivery is called **software maintenance** [Som04, p. 492]. The three different types of software maintenance are adaptive maintenance, corrective maintenance and perfective maintenance [Som04],[LL06]. They correspond to changing requirements, fixing errors and improving the product qualities.

According to Sommerville, "**configuration management (CM)** is the development and use of standards and procedures for managing an evolving software system" [Som04, p. 690]. This means it is part of the maintenance activities. It is concerned with managing different versions of the development artifacts, building systems from these artifacts and dealing with changes in the software.

2.1 Change Request Management

Change request management (CRM) is a software maintenance activity that is part of the configuration management. It is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components of the system have been changed [Som04, p. 696].

A **software problem report (SPR)** is a report that is created when the user has problems using a software system [LL06]. The problem is documented in this report, including the used software version, a problem description and the name of the user to whom the problem appeared.

A **change request (CR)** is a software problem report that requires the software to be changed. These are usually error corrections (corrective maintenance), new requirements (adaptive maintenance) or restructuring the system to improve maintenability (perfective maintenance). Software problem reports that were made due to wrong system usage do not result in change requests.

A change request has usually the properties of a software problem report and some additional properties. The additional properties include a target version of the software that should include the change and additional workflow states like "verified".

Change requests are typically administered in electronic systems called **change request management tools** [LL06]. One common system used for this purpose is Bugzilla [BUGa]. Although there are some products available for CRM, individual solutions are common in companies, too. Bugzilla and some other tools are evaluated in Chapter 3.

2.2 Software Metrics

Fenton and Pfleeger define **measurement** in the following way: "Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules" [FP96, p. 5].

Measurements can be objective or subjective. **Objective measurements** are calculated by a precise, repeatable procedure, whereas **subjective measurements** are based directly or indirectly on individual estimates.

The same measurements over different entities should be comparable. Therefore, measurements have associated scales that define the relationships between the measurement results. There are five different **scale types**: nominal, ordinal, interval, ratio and absolute scales [LL06, p. 19]. These scale types differ in the extent to which they allow comparison of and calculation with the measurement results. The choice of the scale type is directly connected to the measurement itself, for example, one should not use an ordered scale in a measurement of an attribute that has no meaningful order.

Software metrics are special measurements. According to Sommerville, "a **software metric** is any type of measurement that relates to a software system, process or related documentation" [Som04, p. 655]. The entities of which attributes are measured in software metrics are therefore the software product, the software development process and related artifacts and resources. Change requests and software problem reports in particular are software artifacts that can be measured by software metrics.

Metrics can also be derived, which means that they are calculated based on the results of other metrics. Such metrics are called **indirect metrics**, whereas all metrics that are simple measurements are called **direct metrics**.

An ideal metric should posses the following qualities: differentiated, comparable, reproducible, available, relevant, cost-effective, plausible [LL06, p. 281]. It depends on the metric to which extent these qualities are available. Objective measurements are more likely to be differentiated, comparable and reproducible, whereas subjective measurements are more likely to be relevant and plausible [LL06, p. 288].

The goal of software metrics are characterizing, evaluating, predicting and improving [PGF96, p. 3]. In the case of change requests, this is related to the software development process, the software products and the used resources, because the change requests provide some information about all of them.

Although the use of metrics can provide advantages, one should be aware that it takes some effort to measure them and that they might not be as significant as they are expected to be. If the performance of developers and teams is evaluated based on metrics, the developers are tempted to manipulate the metric results by applying a style of work that improves their metric results. Because such a behavior turns the metrics worthless, one should be aware of performance evaluations based on metrics [DeM95].

2.3 Goal-Question-Metric (GQM) Approach

A common approach for defining software metrics is the **goal-question-metric (GQM)** approach [RU89]. It is based on the idea that measurements should be used to achieve higher-level goals which depend on the context of the organization that uses them. This is contrary to a random use of metrics defined in the literature, which causes costs with questionable results. Rombach describes GQM as follows: "The GQM paradigm provides a mechanism for specifying measurement goals and refining them into a set of quantifiable questions and metrics in a tractable way" [RU89, p. 582].

In the first step of GQM, **measurement goals** are defined. They should be on a lower level than business goals [PGF96]. Measurement goals have an object of interest, a purpose, a perspective, and an environment. The object is the entity that should be described with measured values. The purpose is the reason why the measurement activity should take place. The perspective describes who is interested in the metric results. Finally, the environment describes the context for interpreting the metric results.

In the second step, the goals are translated into **quantifiable questions**. Answering these questions should provide valuable information about the previously defined measurement goals. Finally, concrete "metrics are derived to describe what data might be collected to answer the questions" [RU89, p. 583]. A metric can be related to more than one question and answering one question can depend on multiple metrics.

Rombach provided a measurement model for the GQM approach. It can be outlined as follows [RU89]:

1. Specify measurement goals. These are the first steps of GQM described above.

2. Plan measurement process. This includes stating hypothesis about the current state and planning how the process will be executed.

3. Perform measurement process. Performing the measurement process means collecting and validating the data.

4. Interpret collected data in the context of the specified measurement goals. In this step, the data is analyzed and the hypothesis from the planning step are validated.

The GQM approach was further improved by Park, Goethert and Florac [PGF96], who created the **GQ(I)M approach**. The 'I' in GQ(I)M stands for indicator. Indicators in this context are representations for the data from the metrics, for example information graphics or special table representations. The advantage of this is the clear separation of the data itself and the way it is represented to the user.

2.4 Charts

"**Data graphics** visually display measured quantities by means of the combined use of points, lines, a coordinate system, numbers, symbols, words, shading and color" [Tuf98].

A **chart** is a special kind of data graphic. It displays the result of a function that is parametrized with one or more parameters. The maximum number of parameters is determined by the type of chart, the most common case are two parameters. The values that are used as input for each parameter are fixed, and each combination of values is used as input if there are different parameters. The co-domain of the function must be an interval, ratio or absolute scale.

Because measurements can be regarded as functions, their results can be used as input for charts if their scale is applicable for charting. The different parameters could be entities and time points. Another possibility is a meta-function were one parameter is the concrete measurement function and the other parameters are the same as above. Thus, one or more measurements can be displayed in a chart.

Common attributes of charts are a title and a legend. The legend explains what parameter values are mapped to which parts of the chart.

In the following, the chart types that are relevant for this thesis are discussed.

A **pie chart** (see Figure 2.1) is a chart that displays the result of a function with one parameter in a filled circle that is divided in different pieces. Each parameter value is mapped to one piece. The size of that piece is determined by the ratio of the result for that parameter value in the sum of all results.

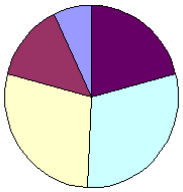

Figure 2.1: *Pie chart*

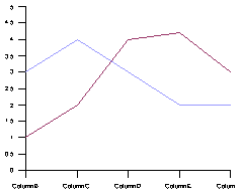

Figure 2.2: *Line chart*

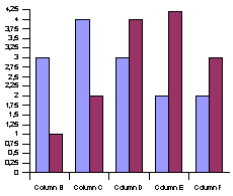

Figure 2.3: *Bar chart*

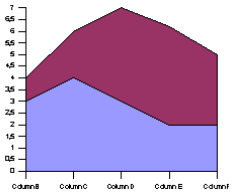

Figure 2.4: *Stacked area chart*

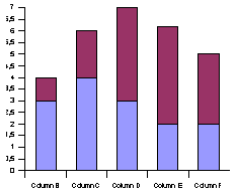

Figure 2.5: *Stacked bar chart*

Because a pie chart calculates ratios, it requires that the co-domain of the function is a ratio scale.

The following charts have a **domain axis**, which is the horizontal axis, and a **range axis**, which is the vertical axis. The range of values on the domain axis are determined by the parameters of the function. The range of values on the range axis is determined by the range of results of the function. The axis are labeled with the corresponding values.

Line charts (see Figure 2.2) display the results of a function with two parameters. One parameter is used to determine the range of values on the domain axis, and the other parameter determines the different lines. The result of the function is displayed as one vertex of a line. The line and the horizontal position of the vertex are determined by the parameters of the function. The vertical position of the vertex is determined by the result of the function.

Bar charts (see Figure 2.3) display the results of a function with two parameters. One parameter is used to determine the range of values on the domain axis in terms of different cells, and the other parameter determines the position of the bars in each cell. The result of the function is displayed as the height of a bar.

Stacked charts are a modification of the previous charts. In stacked charts, the values for the parameter that determines the different lines respectively the position of the bars are ordered. The position on the vertical axis is relative to the position of the previous parameter value in that order instead of being absolute. For example, if the result for the first parameter value is three and for the second parameter it is four with the parameter that determines the position on the horizontal axis being fixed, the value range on the vertical axis for the second parameter is the range from three to seven.

Stacked area charts (see Figure 2.4) are a stacked modification of line charts. The area beneath the lines is filled and they are stacked above each other.

Stacked bar charts (see Figure 2.5) are a stacked modification of bar charts. Instead of using different bars in a cell, the bars are stacked on top of each other.

The last four chart types can serve as **time series charts**. Time series charts, also called run charts [Kan95, p. 129], are characterized by using an ordered set of time points as the parameter that determines the values on the horizontal axis.

Chapter 3

Evaluation of Existing Tools

During requirements engineering, the analysis modules of the following existing CRM tools were evaluated and compared:

- Bugzilla
- JIRA
- Polarion
- Code Beamer

The goal of the evaluation was the creation of a list of features that should be included in the product. Therefore, the relevant features found in the tools above were listed and classified. The result is used as input for the requirements specification (see Chapter 4).

Evaluated Features

For each CRM tool, the following sets of features were examined in detail:

- Change request search. The possibilities and limitations of searching a set of change requests according to some criteria are discussed in this subsection. This is important with respect to the design of filters for the basic set of change requests in the product.
- Charts. The properties of the different charts available in the tool are evaluated in this subsection. This is important with respect to the charting facility of the product.

- Reports. In this subsection, the relevant reports that are available or can be configured and their limitations are discussed. This evaluation is relevant for the identification of the metrics that should be supported by the product.

In the summary, the features from these tools are summarized to give a general overview.

3.1 Bugzilla

Bugzilla [BUGa] is an open source issue tracker developed by The Mozilla Organization [MOZ]. Version 2.18.6 was evaluated. A modified Bugzilla installation is in use at Kisters AG. In this section, bug is used as synonym for change request, according to the Bugzilla terminology.

The Bugzilla Reporting and Charting Kitchen is the reporting part of Bugzilla. It contains several modules for different types of charts: The graphical reports, new charts and old charts module. These modules are explained in detail in the reports section.

Change Request Search

Bugzilla provides three different levels of search functionality. These are search by keywords, advanced search and advanced search using boolean charts.

The simplest kind of search is the **search by keywords**. The bug status and the product can be selected and keywords can be specified. In the chosen product, bugs with the specified status that contain the keywords in their description or comments are searched and returned.

Additionally, there is an **advanced search** that combines the other two levels of functionality.

A complex mask allows the selection of the product, component, version, target, status, resolution, severity, priority, hardware and operating system for the search. Furthermore, keywords that should be found can be specified for the following bug properties: summary, comment, URL, whiteboard and bugs keywords. The bug owner, reporter, QA contact, CC list members and bug commenters can be searched by keywords, too. Other advanced search mask features contain the search for bugs where a certain property changed

to a certain value in a certain time period, the search for bugs with a certain number of votes, and the restriction of the search range to a list of bugs specified by their identifiers.

The selections in the advanced search mask can be combined with the third level of search functionality, **advanced searching using boolean charts**. The user can specify boolean queries on the different bug properties that can be combined using `and` and `or`. This level of search allows the following fields to be searched in addition to the fields from the advanced search mask: bugs the searched bug should depend on, bugs that should depend on the searched bug, attachments, flags, estimated and remaining hours, hours worked, completeness percentage and some other minor properties.

For each of these boolean queries, an operation can be selected and a set of values the property must match according to the chosen operation can be specified. The following sets of operations are available:

- Equality operations. These are operations that check the equality of the property values to the specified values.
- Containment operations. These are operations that check whether the property values contain the specified values in some way. This includes the containment of regular expressions.
- Comparison operations. These are operations that compare the specified values to the values of the property, assuming that it is an ordered property like integers.
- Change operations. These are operations that check the timespan where a property change occurred, or the values from or to which the property changed.

With the third level of search functionality, a large amount of different searches can be specified.

The search result sort order can be specified, too, as one of the following values: bug number, importance, assignee and last changed.

Charts

There are different types of charts used in the different modules of the Bugzilla Charting Kitchen:

- Line charts. Line charts are used in the new charts, old charts and graphical reports module. They can display time series data or data based on categories. For example, the number of open bugs over time can be displayed. An example for data based on categories is a distribution of bugs on different product components. The lines have different colors and are explained in a legend.

- Stacked area charts. In the graphical reports and the new charts module, it is possible to create stacked area charts instead of line charts. The different areas have different colors and are explained in a legend.

- Bar charts. The graphical reports module allows the creation of bar and stack bar charts. The different bars and bar parts have different colors and are explained in a legend.

- Pie charts. In the graphical reports module, the creation of pie charts is possible. The names for the pieces are displayed in the chart, there is no legend.

All charts can be resized horizontally and vertically.

Reports

The Reporting and Charting Kitchen supports different types of reports. The first group contains reports about the current state of the bugs in the database:

- Tabular reports. The results of the evaluation of a complex search can be grouped in three dimensions. The dimensions are displayed as rows and columns of a table and as different tables.

  Each dimension can have a different grouping parameter, which is a bug property with a finite number of values. Possible grouping parameters are product, component, version, hardware, operating system, status, resolution, severity, priority, target milestone, assignee, reporter, QA contact and votes.

  In each cell of the resulting tables, the number of the bugs that are contained in the base filter and match all three grouping parameter values of that cell is shown. It is linked to a page that contains a detailed list of bugs.

- Graphical reports. The graphical reports module shows the results of the evaluation of a complex search in a chart. Depending on the type of chart that is selected, there are different configuration possibilities:

  - Line and bar charts. Grouping parameters can be specified for splitting the search results into different categories on the horizontal axis, into different lines and bars, and into multiple charts. The range on the vertical axis reflects the different group sizes. The number of bugs that matches a category determines the height of a bar or the position of a point in the line. The chart can be stacked or not. If line charts should be stacked, a stacked area chart is used.
  - Pie charts. For pie charts, only the grouping parameters for splitting the search results into multiple charts and for dividing the sections in a pie chart can be selected. The size of the different pieces is determined by the relative amount of bugs that fall into the category of the piece.

  The resulting charts can be resized and a table or csv view is available.

The other group contains reports about change over time:

- Old charts. The old charts module shows the number of bugs in the specified state like *new* over time using a line chart. As the base set of bugs, one or all products can be chosen. The time period on the horizontal axis is determined automatically. The functionality of the old charts is contained in the new charts, too.

- New charts. The new charts module is much more flexible and allows various types of charts to be created. The basic principle of the charting in the new charts module is the following:

  - Different searches, called data sets, are specified using the advanced search mask. These data sets can be stored and reused in different charts.
  - Multiple of these data sets can be added to a chart. For each data set, the number of bugs that match the filter over time is calculated.
  - Data sets can be displayed in the chart as lines or can be summed up to combined data sets which can be displayed as lines, too.
  - All data sets can be added to a total line. Alternatively, the data sets can be displayed in a stacked area chart instead of a line chart.
  - The time period that is evaluated can be chosen and the resulting chart can be resized.

The graphical reports module is very flexible for creating reports about current state of the bugs in the database. Its limitations are the following:

- The grouping of the different values is not flexible regarding the group content. The selected bugs are split into groups according to some bug property, but only with one group per bug property value. Different bug property values can not be unified in one group while splitting on that bug property.

- There is no support for a weighted calculation of the size of an individual bug. All bugs are counted with a one when determining the size of a group.

- Reports about derived values like the minimum, maximum, average and median are not supported, the only value available for a group is its size.

The new charts module, that is used for time series charts, is very flexible, too. Although it may seem as if it would support all relevant time series based charts by using data sets, there are several limitations, too:

- The main limitation is the fact that only the number of bugs can be calculated. This means that the following calculation types are not possible:

  - Derived calculations where the results of the data sets are used in further operations like dividing, multiplication and determining average, minimum and maximum values. Such operations are useful for advanced metrics. The only possible operation to combine data sets is summing.
  - Calculations that count the number of certain events for bugs. Such events are for example bug state changes.
  - Calculations that measure the length of time periods between certain events.

- There is no support for a weighted calculation of the size of an individual bug. All bugs are counted with a one when determining the size of a data set.

- The time granularity cannot be selected. This can result in less significant results if the daily values vary too much.

3.2 JIRA

JIRA [JIR] is a commercial issue tracker developed by Atlassian Software Systems [ATL]. Version 3.7 was evaluated. In this section, the JIRA term issue is used as synonym for change request.

Change Request Search

The filters in JIRA can be configured with several parameters. The results for the selected values in each parameter are combined by a union operation, and the results for the different parameters are combined by a cut operation. The following parameters can be searched:

- Project. A list of possible projects is available for the selection of one or multiple projects.
- Issue Type. Possible issue types are bug, improvement, new feature, support request, task and sub-task.
- Target version. The target version is the version for which the issues should be fixed. A list of all available target versions for the selected projects is shown here.
- Component. Components the issues must be in can be specified. The components from the selected projects are shown.
- Versions affected by the issues.
- A flexible text search in the summary, description, comments and environment fields of an issue. The query can contain boolean operators and wildcards.
- Reporter. The reporter can be a single user, a group of users, or no reporter.
- Assignee. The assignee can be a single user, a group of users, or the special value unassigned.
- Status. One or more of the available issue states can be selected. The issue states that are available depend on the workflow that is used.
- Resolutions. One or more of the available resolutions for an issue can be selected.

- Priorities. One or more of the available priorities for an issue can be selected.
- Created in time period. A time period in which the issues have been created can be specified. The time period can be open at one end.
- Updated in time period. A time period in which the issues have been updated can be specified. The time period can be open at one end.
- Due in time period. A time period in which the issues will be due can be specified. The time period can be open at one end.
- Resolution in time period. A time period in which the issues have been resolved can be specified. The time period can be open at one end.
- Actual vs. Estimated Work Ratio. An interval in which the estimated progress should be in can be specified. The interval can be open at one end.
- List of issues. A list of identifiers for issues that should be returned can be added.
- Participants. Participants of the selected issues can be specified.

The searches can be used to determine the base issue set for reports.

Charts

There are different types of charts in JIRA, line charts, bar charts and pie charts. The type of the chart is fixed per report and depends on the selected report.

The line charts are time series charts. They have the time values on the x-axis. There are different lines with different colors for different types of values. The area between the different lines can be colored, for example in the created vs. resolved issues reports. Then, the area between the lines is colored in the color of the upper line.

The bar charts are time series charts, too. Some of the bars are split to display different values and their sum, for example in the recently created issues report. Then, the different parts of the bars are colored differently.

For line and bar charts, the range of the vertical axis is calculated automatically, depending on the available range of values. For different project

versions, markers on the domain axis can be displayed to show the release date and the version in the chart.

The main colors used in the line and bar charts are green and red, with their common connotations. Green is used for data sets where high values are desirable, whereas red is used for data sets where low values are desirable.

The pie charts only show a distribution of a grouping of the current state of issues. The pie sizes are the relative sizes of the groups. The colors of the parts are different. The description and the absolute size of a piece is shown in a note for each piece.

For all the above charts, in addition to the chart itself, a data table with the values is available.

Another type of chart used in JIRA is a progress bar, which is a stacked horizontal bar chart with a single bar of fixed size and two parts. These parts are colored green and red, with green marking the progress and red the amount of work thats still left.

Reports

The reports in JIRA can be configured in some or all the following general properties, depending on the report:

- Basic filter. A search that defines the basic set of issues that will be used in the chart.
- Time period. The length of the time periods that are displayed in the chart. Possible values are hourly, daily, weekly, monthly, quarterly and yearly.
- Days previously. The number of days back from now that should be included in the chart.
- Cumulative totals. Defines whether the individual values are added to totals or not.
- Display versions. Defines which versions should be marked on the x-axis of the chart. Possible values are all versions, only major versions and no versions.

The following relevant reports are supported by JIRA:

- Recently created issues. Counts the number of the created and resolved and the created, but yet unresolved issues. The report can be configured by basic filter, time period and day previously. The chart is a stacked bar chart with a bar containing the number of resolved and unresolved issues created in each time period.

- Created vs. resolved issues. Counts the created and resolved issues according to the basic filter. The report can be configured by basic filter, time period, day previously, cumulative totals and display versions. The chart is a line chart with one line for the created and one line for the resolved issues. The area between the two lines is colored green or red, depending on whether there are more resolved or more created issues in the corresponding section.

- Resolution time. Calculates the number of issues resolved in the time period and the total and average resolution time for these issues. The report can be configured by basic filter, time period and day previously. The chart is a bar chart with a bar for the average resolution time in each time period.

- Average age. Calculates the number of unresolved issues and the total and the average age time of these issues. The report can be configured by basic filter, time period and day previously. The chart is a bar chart with a bar for the average age in each time period.

- Pie chart. A pie chart report splits the issues selected by the base filter according to some issue property and displays the result in a pie chart and a table. Possible splitting parameters are all available issue properties with finite values.

- User workload report. This report shows a table for the current workload of a user including the number of unresolved issues on a per projects basis.

- Version workload report. The version workload report is based on a certain version of a project. It shows a table with the issue types as columns and users as rows. In the cells, the estimated remaining workload for the user on issues with that issue type in the selected project and version is shown. Furthermore, the estimated remaining workload for each user and issue type is summed up. For each user, a detailed list of the remaining open issues and their remaining estimated effort is shown, too.

- Time tracking report. The time tracking report shows a table that compares the estimated time per issue with the spent time per issue. It contains an accuracy measurement and a progress bar. All values are summed to calculate the estimated and spent time for all issues in the basic filter.

- Single level group by report. This report splits the issues in the base filter according to the selected grouping criterion and shows the groups and their issues in a table. For each group the progress is displayed in a progress bar that renders the percentage of resolved issues.

JIRA has the advantage of supporting age based and time tracking reports, in contrast to Bugzilla. On the other hand, the reports in JIRA are not as configureable as in the Bugzilla new charts module with the data sets. The number of available reports is rather fixed.

3.3 Polarion

Polarion for Subversion [POLa] is a commercial application lifecycle management tool developed by the Polarion Software GmbH [POLb]. It integrates requirements, change and project management tools and provides real-time visibility of the development status. Version 2.6.0 of Polarion was evaluated. In this section, the Polarion term "work item" is used as a synonym for "change request".

Change Request Search

Polarion uses query strings for searching work items. The language for the query strings is the same as that in Apache Lucene [APA], because Polarion uses that search engine.

Every field of a work item can be searched. The searches for the different fields can be combined using `AND`, `OR` and `NOT`. The following types of searches inside a field are supported:

- Simple text or value search. One or multiple values that should be searched for in a field can be specified. Values that must be found can be prefixed with the `+` operator and values that must not be found can be prefixed with the `-` operator. If a text that contains multiple words in a fixed order is searched, it can be written in quotation marks.

- Wildcard search. `*` and `?` can be used as wildcards with the common meaning in strings that are searched.
- Fuzzy search. Fuzzy searches based on the Levenshtein Distance [LEV] are supported. When searching for strings, work items that contain similar strings in that field are found, too.
- Proximity search. For two words, resulting work items can be searched where distance between these two words in the text is less than the specified value.
- Range search. Dates and other values that are inside a specified range can be searched. The range queries can be inclusive or exclusive.
- Grouping. Terms can be grouped by using parenthesis.

The query syntax provides a powerful and flexible solution for specifying searches. A query builder is integrated in Polarion that simplifies the construction of common queries.

Charts

The charting possibilities in Polarion are, compared to other tools, rather limited. Two different types of charts are used:

- Line charts. Line charts are used to show time series data in Polarion. The domain axis contains the time period represented in the chart, and the range axis the range of available values. The different lines have different colors.
- Single stacked bar charts. A single bar of fixed length is split into several parts according to the relative size of the work item group that is represented by the part. This type of chart is compareable to a pie chart, but it is represented as a single bar. The parts have different colors, each green or red, using different shades of green for the parts on the left side and different shades of red for the parts on the right side. If there are only two parts, this type of chart is equivalent to a progress bar.

Reports

The following relevant reports that are supported by Polarion:

- Work items trend. This line chart shows a time series for the new and open work items and the unresolved requirements. The last 30 days are shown on the domain axis, and the number of items is shown on the range axis.

- Unresolved work items by priority. The unresolved work items in a project are grouped by their priority. The size of each group is calculated and rendered in a single stacked bar chart.

- Unresolved work items by severity. The unresolved work items in a project are grouped by their severity. The size of each group is calculated and rendered in a single stacked bar chart.

- Resolved work items by severity. The resolved work items in a project are grouped by their severity. The size of each group is calculated and rendered in a single stacked bar chart.

- Resolved vs. unresolved work items. The number of the resolved and the number of the unresolved work items displayed in a single stacked bar chart.

- Assigned vs. unassigned work items. The number of the assigned and the number of the unassigned work items displayed in a single stacked bar chart.

- Scheduled vs. unscheduled work items. The number of the scheduled and the number of the unscheduled work items that are unresolved displayed in a single stacked bar chart. Scheduled means in this context that there is a time point when the work on that work item should start.

- Estimated vs. not estimated work items. The number of work items with an initial estimate and at least one work time entry compared to the number of all work items is displayed in a single stacked bar chart.

- In time vs. underestimated work items. In the set of estimated work items, the rate of the work items where the initial estimation is greater than the worked time and the remaining estimate is displayed in a single stacked bar chart.

- Project plan accuracy. The project plan accuracy is the overall accuracy of a project. The initial estimates, spent times and remaining estimates for all work items in the project with an initial estimate and at least one spent time or a remaining estimate are summed up. Depending on whether the project initial estimate is greater than the

remaining estimate plus the spent times, it divides the remaining estimate plus the spent times or is divided by them. The result is the overall project plan accuracy that is displayed in a single stacked bar chart.

- Estimate accuracy. Estimate accuracy is the accuracy of the individual work item estimates. Only work items in the project that have an initial estimate and at least one spent worktime entry or a remaining estimate are taken into account. It is calculated as the sum of the initial estimates divided by $estimatedAccuracy = |remainingEstimate + spentTime - initialEstimate|$ and displayed in a stacked bar chart.

- Not suspected vs. suspected work items. This is the rate of unresolved issues that are linked to other work items. The rate is displayed in a stack bar chart.

- Traceability of commits to work items. The rate of commits that are linked to work items is displayed in a single stacked bar chart.

- Traceability work items to commits. The rate of work items which can be traced to a certain repository version. It is displayed in a single stacked bar chart.

Most of the Polarion reports are current state reports represented in stacked bar charts. The support for time series reports is limited. Advantages of Polarion are the traceability, estimation and accuracy reports. On the other hand, the reports in Polarion are not as configureable as in the Bugzilla new charts module with the data sets. The number of available reports is rather fixed.

3.4 Code Beamer

Code Beamer [COD] is a commercial collaborative development platform with application life cycle management features developed by Intland GmbH [INT]. Code Beamer 4.2.1 was evaluated. In this section, the Code Beamer term “tracker item” is used as a synonym for “change request”. A tracker is a collection of tracker items.

Change Request Search

Code Beamer supports search and filtering at different points of the software.

The main search uses the Apache Lucene [APA] search engine. It supports the following options for searching tracker items:

- Keywords. An advanced keyword search for searching tracker items is provided. The searches for the different fields can be combined using `AND`, `OR` and `NOT`. For one field, it supports the following search features:
  - Simple keyword search. A term consisting of a keyword is searched in the tracker items.
  - Required operator `+`. Any term prefixed by the requirement operator must occur in the resulting tracker items.
  - Grouping. Terms can be grouped by using parenthesis.
  - Wildcards searches. The wildcards "?" and "*" can be included in the keywords in the usual way.
  - Fuzzy searches. Fuzzy searches based on the Levenshtein Distance [LEV] are supported for finding tracker items containing words which are similar to the specified keywords.
- Restriction on projects. The search can be restricted to a set of projects.
- Owner/Submitter. The search can be restricted to the tracker items that are owned or submitted by the specified person.
- Created/Submitted. The time period the search tracker items were created in can be specified.
- Modified. The time period the search tracker items were modified in can be specified.

Code Beamer supports variable trackers for the project. Each tracker can have its own fields, which makes searching more complicated. To solve the problem of filtering trackers, Code Beamer supports customer tracker views. They are configured with a name, filtering, displayed fields and sorting. The filtering relevant here can be configured as follows:

- For fields with a fixed, finite set of values, a subset of these values can be selected for filtering. It can be combined with a not that selects the inverse set. This is applicable for fields like priority or status.
- Fields of a number, text or data type can be searched by comparison operations. The field, the operation, and the searched values are selected and form a search criterion. The following types of operations are available:

  - Equality operations. Equality operations compare if the value in the field of the tracker items equals the specified values. The non-equality operation is available, too.
  - Containment operations. Containment operations check whether the specified keywords are contained in the tracker item field or not. The starts-with operation is also available.
  - Comparison operations. Comparison operations compare if the specified value is greater or less than the tracker item field value.
  - Field value availability check operations. These operations check whether the field of the tracker item is set or not.

  The different search criteria can be combined with `AND` and `OR` and they can be grouped.

Code Beamer uses a third variant of tracker item filtering in the specification of tabular reports. Here, in addition to selecting a set of allowed values for fixed, finite field types, an additional SQL `WHERE` statement can be specified to further narrow the search.

Charts

Code Beamer uses two types of charts:

- Bar charts and stacked bar charts. Bar charts and stacked bar charts are used to display information that is split into categories and some simpler time series data.
- Line charts. Line charts are used to display time series data in Code Beamer.

Reports

Code Beamer supports the following relevant reports:

- Last 7 days changes. A stacked bar chart shows the number of submitted, edited and closed tracker items for the last 7 days.
- Last 7 days trend. A line chart shows the number of open tasks and the estimated and spent hours over the last 7 days.
- Bugs by date. The number of total, open and closed bugs for the last two months is displayed in a line chart.

- Bugs by category. The current open and closed bugs are split into their categories, which is used in Code Beamer as a synonym for component. The result is displayed in a stacked bar chart with one bar for each category.

- Bugs by severity. The current open and closed bugs are split into their severities. The result is displayed in a stacked bar chart with one bar for each severity.

- Bug age by severity. The age of the open bugs in weeks is determined. The open bugs are split according to their age in weeks and then these groups are split again by their severities. The result is displayed in a stacked bar chart with one bar for each week of age and with the severities as bar parts.

- Bugs per KSLOC. The number of total bugs detected per thousand source lines of code is displayed in a time series line chart. The evaluated time period is the last two months.

- Tasks by date. The total and open task time series for the last two months is shown in a line chart. The line chart contains a time series for the submitted and edited tasks, too, but these are not summed up, but shown as their daily values.

- Requirements with cumulative changes. This time series line chart shows the number of total and open requirements over the last two months. Additionally, it shows the cumulative changes to requirements in that time period.

- Requirement changes by date. This time series line chart shows the number of total and open requirements over the last two months. Additionally, it shows the number of new and changed requirements on a daily basis over that timespan.

- Estimated and spent task efforts in hours. This time series chart shows the number of estimated and spent task efforts in hours for both the open and the closed tasks. It is shown in a line chart that ranges over the time period of the last two months.

- For the different work item types like bug or feature request, a report by status and severity is available. The work items of the selected type in the project are split into groups by their status and severity and the result is displayed in a bar chart and a table. The severity is used for

the creation of bar groups, whereas the status is used for the colors of the bars in the bar groups. The height of the bars reflects the number of work items with that severity and status.

Code Beamer supports a fixed set of reports, much like Polarion. It lacks the flexible configuration of time series reports available in Bugzilla. An advantage of Code Beamer is the integration of requirements, which supports reports like "requirement changes by date".

3.5 Summary

In this summary section, the main features of all evaluated tools are highlighted.

Change Request Search

The search functionality provided by the different tools ranges from a simple keyword search in a fixed set of properties to the specification of the `WHERE` part of an SQL statement.

Whereas a simple keyword search is not sufficient for selecting the base set of change requests, SQL statement parts are on a level of abstraction that is too low for the use in the final product. Therefore, the intermediate level features are summarized here:

- Text fields can be searched by simple keywords, wildcards, fuzzy and proximity search. Further options for text fields are a starts-with search and an equality search.
- Text fields can be searched by regular expressions.
- Fields of an ordered type like text fields, date fields and number fields can be searched by a range search.
- Fields of an ordered type can be searched by comparison with other values.
- Fields with a finite, fixed set of values can be searched by a set of these values.
- The basic search terms can be grouped and combined with the boolean operators `AND`, `OR` and `NOT`.

- Change requests can be searched by their creation and modification dates, especially by searching change requests with modification or creation dates that are in certain time periods.
- Change request can be searched by their field changes. The original or new value that is searched can be specified for such a filter.

Charts

The charts supported by the tools can be grouped into charts that support the display of time series and those that do not. The charts that cannot be used for time series data, but only for a snapshot like the current state, are the following:

- Pie charts. Pie charts show a distribution of groups of change requests. Normally, a basic set of change requests is split into these groups according to some criterion, for example the available values in a field.
- Horizontal single stacked bar charts. These charts are similar to pie charts because they show a distribution of groups of change requests. But instead of showing that distribution in a pie, it is shown in a horizontal bar. Progress bars are a special form of these charts, because they contain only two pieces.

The other types of charts can be used to display time series data and snapshot data:

- Line charts. Line charts show several data sets in different lines. The domain axis can contain a time period or categories. If the data sets have different ranges, there can be multiple range axes.
- Stacked area charts. Stacked area charts are the stacked form of line charts. Instead of drawing each line independent of each other, they are stacked on each other and the area between them is filled.
- Bar charts. Bar charts have several data sets for different bars. The domain axis can contain a time period or categories. For each entry in the domain axis, one bar from each data set is displayed. If the data sets have different ranges, there can be multiple range axes.
- Stacked bar charts. Stacked bar charts can contain several pieces per bar. The pieces are stacked on top of each other. Usually, there is only one bar for each entry on the domain axis, but it is also possible that another splitting dimension is added and that there are several stacked bars for each entry on the domain axis.

Another important feature is the use of markers on the domain axis to show the date of versions in a time series.

Reports

The first distinction for the reports is the distinction between snapshot and time series reports. Snapshot reports calculate values for a specific point of time, most often the current date. Time series reports calculate values at different time points in a time period. The different time points in the time period are usually calculated by splitting it into time intervals of a fixed duration and using the end points of the intervals as time points.

The relationship between time series reports and snapshot reports is the fact that a time series report can be split in multiple snapshot reports with the same calculation, and vice versa. The relationship can be cumulative, meaning that the snapshots are added, or not. Because of this relationship, it is only necessary to evaluate snapshot reports, as they can be extended into time series reports.

The following types of snapshot reports appear in the evaluated tools:

- Splitting. According to some criteria, the change requests are split into groups. The size of the groups is calculated and displayed in the snapshot report. Splitting includes counting the number of all change request as a special case, namely splitting into one group.

  Examples for these kinds of reports are the number of created change requests, the number of closed change requests, the state distribution of open change requests, the priority distribution of all change request and so on.

  The splitting can be multi-dimensional, for example a splitting into priority and status.

- Age based reports. Age based reports like the resolution time of closed change requests or the average age of open change requests require the calculation of the length of a time period.

- Workload reports. The workload reports are based on the amount of time spent on change request and the original and remaining estimations on how much time must be spent to resolve a change request.

- Linked reports. Reports that link the change request to external artifacts like requirements or the version control system.

- Derived reports. Reports that calculate derived values based on other numbers, like the original effort estimation accuracy.

Reports usually have a basic filter that determines the set of cases that are evaluated. The base filter can be just the selected project, but it can be an advanced filter, too.

Another important parameter is the time granularity. It determines the length of the time period for the snapshot evaluation, for example, that all created change requests on one day should be counted.

For time series reports, there is a time period that should be evaluated, whereas for snapshot reports, there is a time point.

Chapter 4

Requirements

Based on the results from the evaluation of the existing tools (see Chapter 3), the concepts and terms of the software process metric domain were analyzed and a requirements specification was created. This chapter contains the important parts of the requirements specification.

4.1 Introduction

4.1.1 Goal and intention of the product

The tool should evaluate metrics on the data contained in the change request management (CRM) system used by Kisters AG. The resulting information can be used for several purposes:

- Evaluation how the CRM system is used by the employees. This means measuring the quality of the data collected in the CRM system.
- Improvement of the awareness of the current project states. This means measuring the workload and the software product quality.
- Finding software development process weaknesses. This means measuring the software development process quality and speed.
- Measuring whether development process changes improved the process or not. The measurements required for this point are measuring the workload, the product quality and the software development process speed and quality.

4.1.2 Users of the product

The product will be used by different employees at Kisters AG, including project managers, department managers and quality assurance staff.

The different users will be using the product for getting different kinds of information. These kinds of information range from detailed information like the bugs that are overdue in a specific component to general information like the development of the average estimated remaining workload per employee.

4.1.3 Assumptions and dependencies

The CRM system in use at Kisters is a modified Bugzilla 2.18.5 installation running on a MySQL 4 database. The modifications to Bugzilla include

- the introduction of different case types. At Kisters AG, Bugzilla bugs are called cases, and software defects are called bugs. The case types are encoded in the severity field of the case.
- the introduction of additional case fields like deadline and customer. These fields are stored in a rate table.
- changes in the case workflow. New case states like "waiting for customer information" were introduced.

4.2 Concept Maps

4.2.1 Change Request Management

In change request management (CRM, see Figure 4.1), change request management tools are often used to administer change requests. CRM tools are software programs that store change requests in a database and help accessing that data. They are a subclass of issue tracking systems which manage issues. Issues are a generalization of change requests that include support calls and other tasks as well.

Bugzilla is a special bugtracker. Bugtrackers are a special class of issue trackers. It manages bugs, which are used as a synonym for cases and issues.

There are three types of cases at Kisters AG: support calls, bugs and enhancement requests. Only enhancement requests and bugs are real change

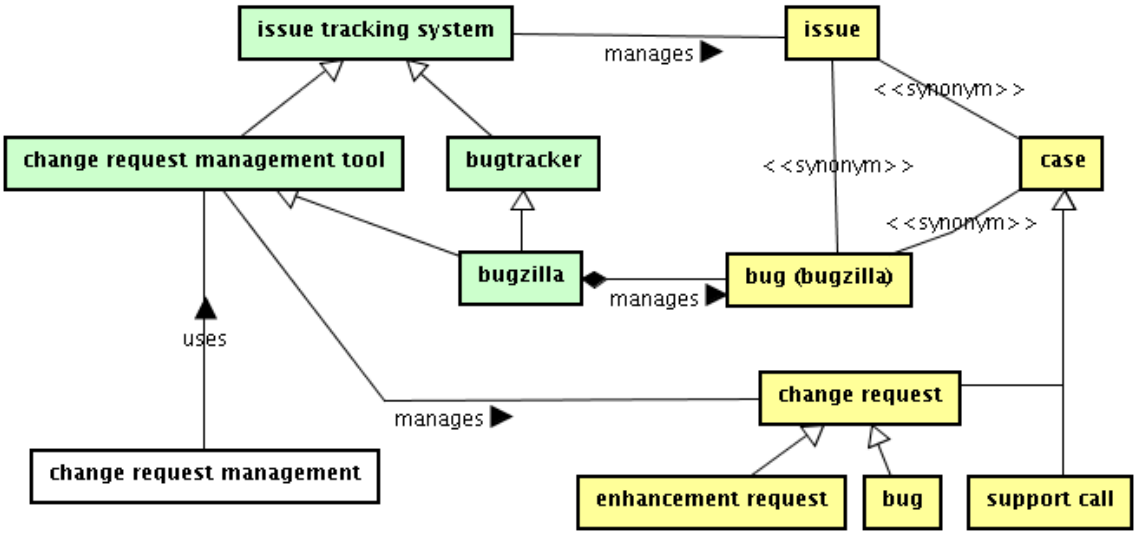

Figure 4.1: *Concept map "change request management"*

requests. Enhancement requests are requests to add new features to a software, whereas bugs are requests to fix defects in a software. Support calls fall into the class of software problem reports, because they do not necessarily lead to a change of the software.

4.2.2 Case

A case (see Figure 4.2) is a synonym for an issue. It has a case type, which can be a bug, an enhancement request or a support call. Other properties are:

- Deadline. The date on which a case has to be verified and shipped.
- Severity. The severity is an integer between 1 (highest) and 4 (lowest) that indicates how important the work on a case is to its customer.
- Priority. The priority is an integer between 1 (highest) and 4 (lowest) that indicates how important the work on a case is to Kisters AG.
- Actual effort. The number of hours spent working on the case.
- Estimated effort. An estimation of how much work in hours will be required to resolve the case.

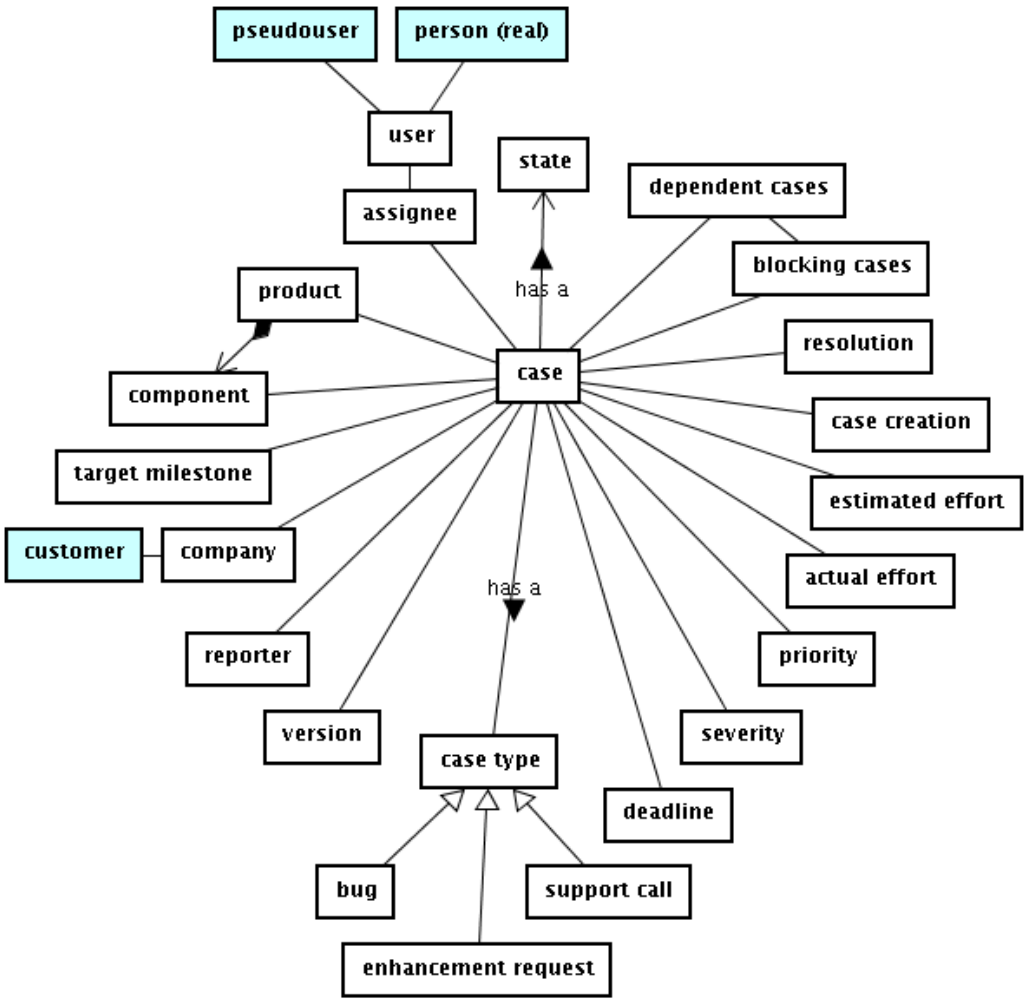

Figure 4.2: *Concept map "case"*

- Case creation timestamp. The time and date at which the case was created.
- Resolution. The way a case was resolved, for example fixed or duplicate.
- Blocking cases. Cases that need to be resolved before work on this case can proceed.
- Dependent cases. Cases that need this case to be resolved before work on them can proceed.
- Case state. The state of work the case is currently in. It is described in detail in the next section.
- Assignee. The Bugzilla user that is assigned to work on the case. It can be a real person or a pseudouser which stands for a certain class of employees like teams.
- Product. The software product the case belongs to. A product consists of different components.
- Component. The software component the case belongs to. A component belongs to a certain product.
- Target milestone. The milestone of the product that should contain the bugfix for a bug case.
- Company. The company the work on the case is done for. It can be a customer, but it can be Kisters AG itself or a customer group, too.
- Reporter. The person who entered the case into the Bugzilla database.
- Version. The version of the software the case was reported against.

4.2.3 Case State

A case state (see Figure 4.3) is the state of work the case is currently in. There are nine different states:

- New. The state of a case that has just been created, but is not yet assigned.
- Reopened. The state of a case when the QA department or the customer has discovered that the case was not resolved as expected and further work needs to be done before it can be resolved again, but it is not yet assigned.

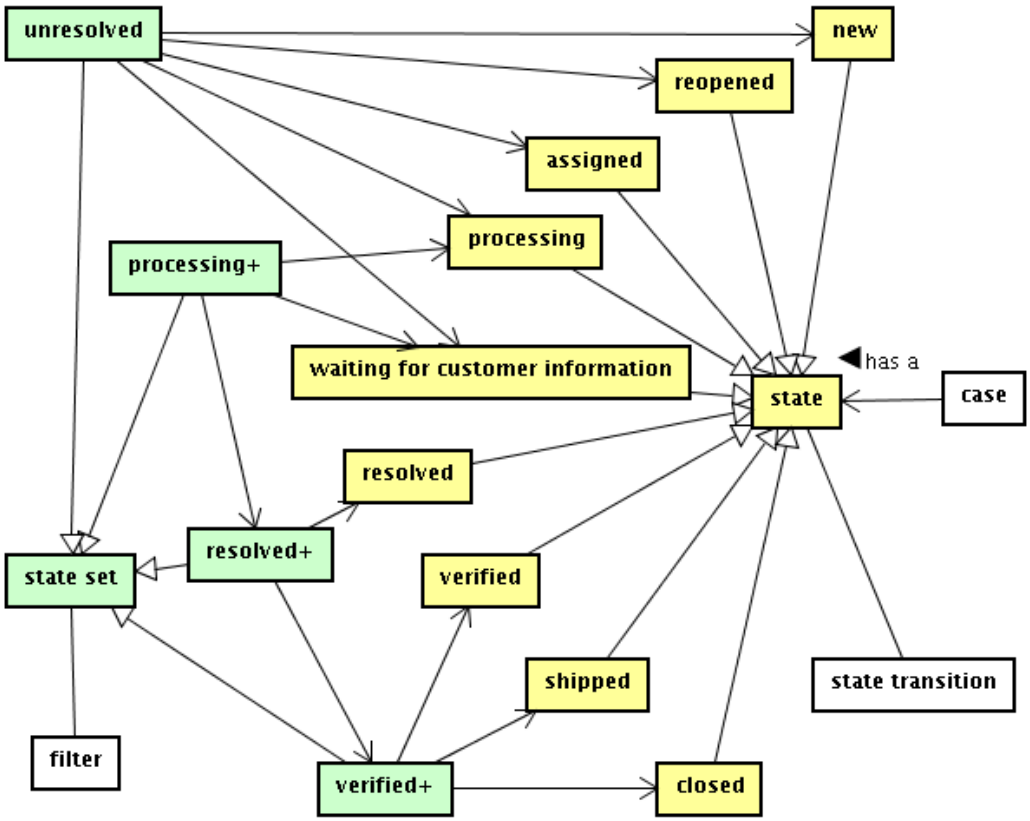

Figure 4.3: *Concept map "case state"*

- Assigned. The state when an assignee has accepted a case, but is currently not working on it.
- Processing. The state of a case when its assignee is currently working on the case.
- Waiting for customer information. The state of a case when the assignee needs and has asked for further information from the customer, but the customer has not yet delivered that information.
- Resolved. The state of a case when the assignee has finished working on the case and the QA department has to verify that the work was finished correctly.
- Verified. The state of a case when QA department has verified that the assignee has resolved it correctly.
- Shipped. The state of a case when the software that includes the result of the work on that case has been delivered to the customer.
- Closed. The state of a case when it has been shipped to the customer and the customer has reported that everything works as expected.

A state transition is the change from one state to another. Not all possible transitions between states are allowed, for example, once a case has been resolved, it cannot be directly changed to assigned, it has to be reopened first.

Case states can be grouped into state sets. Examples for state sets are:

- Unresolved. Set of states when a case has yet to be resolved. Includes new, reopened, assigned, processing and waiting for customer information.
- Processing+. Set of states once work on a case has begun. Includes processing, waiting for customer information, resolved, verified, shipped and closed.
- Resolved+. Set of states for resolved cases. Includes resolved, verified, shipped and closed.
- Verified+. Set of states for cases that have been verified. Includes verified, shipped and closed.

State sets can be used to filter cases.

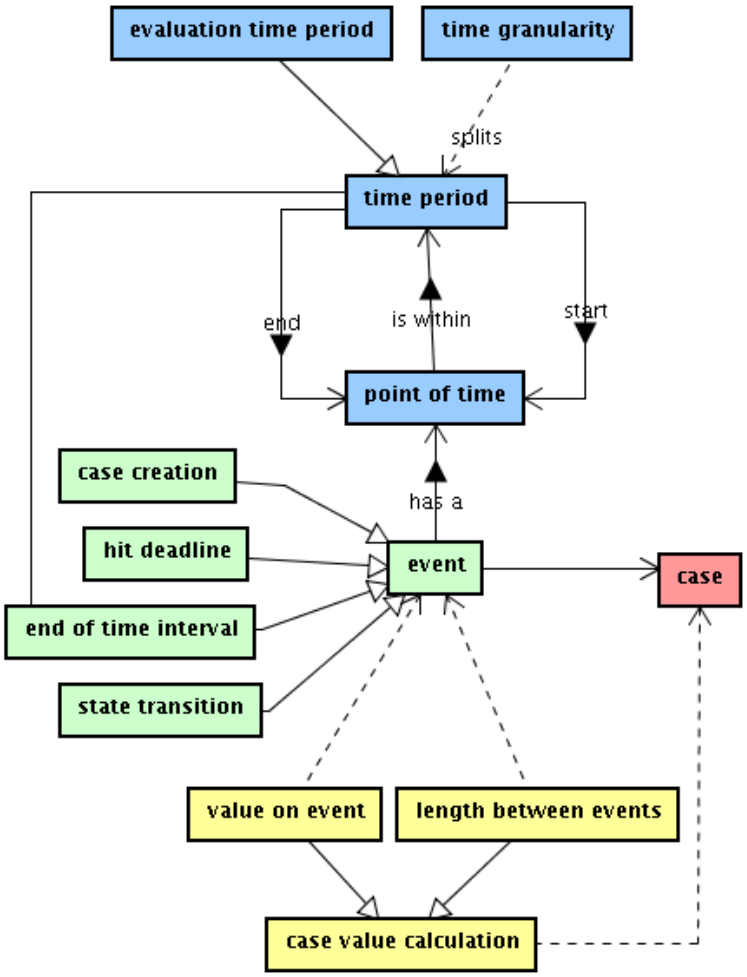

Figure 4.4: *Concept map "event"*

4.2.4 Event

An event (see Figure 4.4) is something that occurs to a case at a given point of time. Examples for events are

- the creation of a case
- a case hits its deadline
- the end of a time interval
- a case state transition

Events are used in the case value calculation (see Section 4.3.1). Different case value calculations like the calculation of a value on an event or the length between two certain events are triggered by events. They assign a value to the event that is used later in the group value calculation.

Each point of time lies within a time period. The start and the end of a time period are itself points of time. The evaluation time period, which is itself a time period, can be split into several time periods by using a time granularity.

4.3 Functional Requirements

Because of the relationship between snapshot reports and time series reports (see Section 3.5), and the fact that the Bugzilla Reporting and Charting Kitchen already supports snapshot reports, the tool should only evaluate time series reports.

4.3.1 Basic Algorithm Parameters

The basic algorithm (see Figure 4.5) can be parametrized in several ways:

- Basic filter. The basic filter determines which cases will be evaluated. It should be configureable to allow restriction to products, assignees, components etc.
- Group parameter. The group parameter determines how the evaluated cases will be grouped into case groups before the group itself is evaluated. Examples are product, component or priority. It should be possible to do a nested grouping, for example group by products on the first level and by their components on the second level.

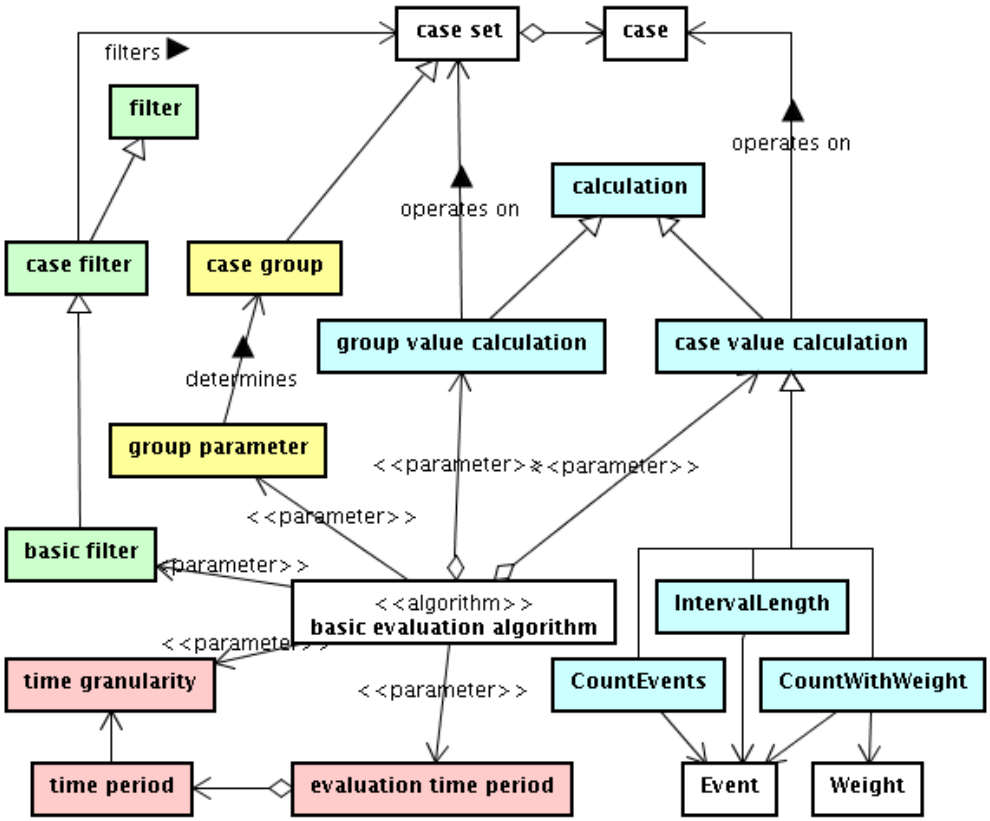

Figure 4.5: *Basic algorithm structure*

- Case value calculation. This parameter determines how the values for a case on a certain event are calculated. The algorithm can be parametrized with more than one case value calculation. Possible types of case value calculations are counting with a weight, calculating time interval lengths and counting event occurrences. Weights are calculations based on the state of cases on a certain event.

- Group value calculation. This parameter determines how the result value for a group is calculated from the results of the case value calculations for the cases in that group. The algorithm can be parametrized with more than one group value calculation.

- Evaluation time period. The time period that should be evaluated.

- Time granularity. This parameter determines how the evaluation time period is split into the time periods for which the group values are calculated. Examples for time granularities are week, month and quarter.

4.3.2 Metrics

In addition to the parameters of the algorithm, several metrics were specified that the product must support. They are described in the following.

- Percentage of resolved cases without actual effort values. Calculates the percentage of cases without actual effort values in the set of all relevant resolved cases.

- Percentage of cases without version value. Calculates the percentage of cases without version value in the set of all relevant cases.

- Percentage of resolved bugs without a target milestone. Calculates the percentage of bugs without a target milestone in the set of all relevant resolved bugs in a time period.

- Original effort estimation accuracy. The original effort estimation accuracy is a measurement how precise the initial effort estimation for a case is compared to the actual effort required to close it. For each case, it is calculated as

$$1 - \min(1, \frac{|\text{originalEstimatedEffort} - \text{actualEffort}|}{\text{originalEstimatedEffort}})$$

- Totals. The totals metric calculates the number of open cases.

- Incoming rate. The incoming rate is the number of cases added during a time period.
- Outgoing rate. The outgoing rate is the number of cases resolved during a time period.
- Average age before first entering state set. The state set must be further specified, for example as "processing+" (see Section 4.2.3).
- Backlog Management Index (BMI). The backlog management index is calculated as follows [Kan95, p. 105]:

$$\text{BMI} = \frac{\text{outgoingRate}}{\text{incomingRate}}$$

- Totals overdue. The totals overdue metric calculates the number of open cases that are beyond their deadline. Cases without a deadline are not taken into account.
- Reopened rate. The reopened rate metric calculates the number of cases reopened during a certain time period.
- Case lifetime. The age of open cases at that timepoint and the age of closed cases at their closing.
- Estimated remaining workload. The estimated remaining workload is calculated as the sum

$$\sum_{\text{case } c} (\text{estimatedEffort}_c - \text{actualEffort}_c)$$

  over all relevant cases c.
- Number of state changes before resolution. This metric counts how often the state of a case was changed before it has been resolved.
- Number of assignee changes before processing. This metric counts how often the assignee of a case was changed before processing that case started.
- State residence time. The state residence time is a measurement how long the cases stay in their different states.
- Customer activity. The customer activity is the amount of cases entered by or for a certain customer.

- Defect rate. The defect rate is the percentage of bugs in the set of open bugs and enhancement requests.
- User activity distribution. The user activity distribution counts how many open bugs, enhancement requests and support calls there are for a user.
- Bug assignment rate. The percentage of bugs compared to bugs and enhancement request in the cases of a user.
- Product work distribution by components. This metric calculates the totals for each component of a product.
- Correlation of Priority and Severity. This metric calculates how the different severities specified by the customer correlate with the different internal priorities.

4.3.3 Metric Categories

The metrics described above can be categorized by the goal that they are used for. Using that criterion, a metric can fall into one or many of the following metric categories:

Quality of collected data. Metrics that measure the quality of the data collected in the Bugzilla database itself fall into this category. Quality in this context means completeness, accuracy and precision of the data. The results of metrics of this category should be considered when interpreting other metrics, because these might be based on incorrect or incomplete data.

Workload. Metrics that measure how much work was done and how much work is still left fall into this category.

Process speed. Metrics that measure how fast cases are processed fall into this category. This includes the overall processing time and the processing time for certain case states.

Process quality. Metrics that measure process qualities like project performance, planning reliability and inner process qualities fall into this category.

Product quality. Metrics that measure the current state of the products fall into this category.

Metric	Quality of collected data	Workload	Process Speed	Process Quality	Product Quality
Percentage of Resolved Cases without Actual Effort Values	√				
Percentage of Resolved Cases without Version Value	√				
Percentage of Resolved Bugs without Target Milestone	√				
Correlation of Priority and Severity	√				
Original Effort Estimation Accuracy	√			√	√
Estimated Remaining Workload		√			
User Activity Distribution		√			
Totals		√			√
Incoming Rate		√			√
Outgoing Rate		√			√
Product Work Distribution By Components		√			√
Average Age Before First Processing		√	√	√	
Backlog Index		√	√	√	√
Reopened Rate		√	√	√	√
Case Lifetime		√	√	√	√
Bug Assignment Rate		√	√	√	√
State Residence Time		√	√	√	√
Number of State Changes Before Resolution			√	√	
Number of Assignee Changes Before Processing			√	√	
Totals Overdue			√	√	√
Customer Activity					√
Defect Rate					√

Figure 4.6: *Metric Categories*

Table 4.6 shows a classification of the metrics from the previous section according to the metric groups described above.

4.3.4 Charts

The product should support the creation of charts for the metric results. The charts should be line charts or stacked area charts, depending on whether they are configured to be stacked or not.

They should be time series charts with the evaluated time period shown on the domain axis. The height of the range axis should be determined automatically depending on the values in the metric result.

The data sets that are displayed in the charts should be selectable by specifying which group calculations should appear.

The legend should be generated automatically, too. It should use the group names and the group calculation names to determine the name for a data set.

Additionally, special configureable timepoints like release dates that are highlighted in the chart should be supported.

4.3.5 Web Frontend

The web frontend should provide an interface for evaluating metrics and displaying them in charts. The usability of the web frontend is less important than the functionality of the core module and the chart module.

4.4 Non-Functional Requirements

4.4.1 Performance Requirements

The system will be accessed through the web interface and should respond within 5 minutes to a query.

The Bugzilla database at Kisters will get larger as more data is entered over time. Therefore, the system should scale when used on a large database. As it is expected that up to ten thousand cases are added per year, the system should work efficiently on up to one hundred thousand cases.

4.4.2 Maintenance Requirements

As the Bugzilla installation at Kisters will change, the product must be changed, too. Therefore it should be easy to change the representation of case fields in the database.

The product should be extensible regarding the evaluation of further metrics.

4.4.3 Security Requirements

The Bugzilla database contains sensitive data. The product must not change this data under any circumstances. Additionally, it must be guaranteed that the product cannot be misused from unauthorized persons to retrieve sensitive information from the database.

4.4.4 Other Requirements

The product should be implemented using the programming language Java and be based on the Java 2 Platform, Standard Edition (J2SE) 1.4.2 [JAV].

The web frontend should be implemented as web application that conforms the Java Servlet Specification 2.3 [SER].

Chapter 5

Software Process

According to Sommerville, "a software process is a set of activities that leads to the production of a software product" [Som04, p. 64] and "a software process model is an abstract representation of a software process" [Som04, p. 65].

The choice of the software process model used in a project has a big impact on the success of the project. Therefore, it is important that it is chosen carefully.

This chapter is organized as follows:

- The project constraints are described.
- The consequences of the project constraints are examined.
- The used software process and how it is related to Extreme Programming (XP) [Bec00] is described.

5.1 Project Constraints

The choice of the software process model was based on constraints that can be grouped into project complexity, infrastructure integration and domain knowledge constraints.

The project complexity constraints are the following:

- The project has one developer. The developer is familiar with agile methods, especially XP.
- The project must be finished at a fixed deadline.

- The product size is small. It is expected that the resulting source code size will be less than ten thousand lines of code.

- The product is non-critical with respect to the definition given by Sommerville [Som04, p. 44]: "If [critical systems] fail to deliver their services as expected then serious problems and significant losses may result". It is neither safety-critical nor mission-critical nor business-critical.

The infrastructure integration constraints are the following:

- The product uses data supplied by the existing tool infrastructure. This tool infrastructure might change.

- The product will be integrated in the existing tool infrastructure. Future tools are expected to use the services provided by the tool.

The domain knowledge constraints are the following:

- Neither the developer nor the customer are familiar with the domain of software process metrics that is explored during the project.

- There are no architecture patterns available for software process metric evaluation systems.

5.2 Project Constraint Consequences

The project constraints have several consequences that influence the choice of the software process model.

Because both the customer and the developer are unfamiliar with the domain, it is likely that new requirements are discovered after parts of the system have been implemented. It is also likely that the priority of the requirements changes and that old requirements are dropped.

Furthermore, because the requirements are likely to change and the developer is not familiar with the domain, the system architecture and design is likely to change, too. Reasons for such changes may be new or changed requirements, discovering of tool dependencies, and a better understanding of the variation points and concepts of the domain.

5.3 Process Description

As highlighted in the previous section, the requirements and the system architecture are likely to change during the first iterations. This is the reason why an agile software process model was chosen. As the developer already has some experience with XP, this software process model was adopted to the circumstances of the project.

The used software process model is divided into three main phases:

1. **Initial requirements engineering**. The goals of this phase are the creation of an initial version of the requirements specification and a basic understanding of the domain concepts and terms.

2. **System implementation**. The goal of this phase is the creation of a software product that fullfills the requirements. It includes several minor rollouts.

3. **System documentation and rollout**. The goals of this phase are the documentation of the software, the final rollout and an increase of the robustness of the software.

Each phase consists of one or many iterations, and in the implementation phase, elements from agile software process models are incorporated. An iteration usually takes between one and one and a half months and is finished by a meeting. This meeting serves as kick-off meeting for the next iteration, too.

The software process model is based on the following principles of agile methods [Som04, p. 397]:

- Customer involvement. At the end of each iteration, the current project state is evaluated and the goals for the next iteration are negotiated with the customer.

- Incremental delivery. In each iteration of the system implementation phase, a new part of the product is developed and evaluated by the customer.

- People not process. The skills of the developer, especially the knowledge of the Java programming language, the Eclipse IDE and the JUnit test framework are exploited.

- Embrace change. The variation points of the system are identified and the system is designed to accommodate changes on these variation points. The requirements and the system architecture are expected to change during the development.

- Maintain simplicity. The system is refactored and restructured in order to keep it as simple as possible. The development process is kept simple, too.

5.3.1 Initial Requirements Engineering

In the requirements engineering phase, the result of each iteration was a version of the requirements specification.

In the first iteration, existing CRM systems were examined with respect to the process metrics and process metric visualizations they provide. The result of this evaluation, which is also available in Chapter 3, was used as base for the discussion with the prospective users.

Furthermore, the domain terms and concepts were identified by creating concept maps. These concept maps were discussed with the prospective users. They were used to create a glossary of the domain terms, which was discussed with the prospective users, too.

After this basic exploration of the domain, the first version of the requirements specification was created. It contained a range of possible process metrics, other functional requirements and the system constraints. This document was discussed with the prospective users and checked for validity, consistency, completeness, realism and verifiability in the meeting at the end of the first iteration.

In the next iteration, the requirements specification was revised according to the results of the validation in the previous meeting. Additionally, the process metric requirements were refined and split into reoccurring components. Based on this restructuring, the first variation points of the evaluation algorithm were discovered and a coarse model of the algorithm was designed.

This second version of the requirements specification was verified, revised again and used as input for the implementation phase. The important parts of the requirements specification are available in Chapter 4.

5.3.2 System Implementation

In each iteration of the system implementation phase, a new part of the system was developed. The goals of the iterations can be summarized as follows:

1. Development of a first version of the core module and implementation of some simple metrics to test the module.

2. Implementation of the remaining metrics and a first test of the system on real data.

3. Optimization of the system on real data and creation of some preliminary charts using a spreadsheet software.

4. Implementation of the charting module and web frontend.

It is important to note that the goals were not planned in advance. At the end of each iteration, the goal for the next iteration was set based on the progress made in the current iteration and the problems that occurred.

In each iteration, these relevant elements from XP [Som04, p. 400] are used among others:

- Incremental planning. The goals for each iteration are defined in its kick-off meeting. The developer breaks down these goals into tasks he executes. In the kick-off meeting of an iteration, the requirements can be changed and re-prioritized by the customer.

- Test-first development on specification level. A test framework (see Section 6.14) was designed that allows tests to be defined in XML documents on specification level. These documents serve both as tests and as reference on how to use the software.

- Refactoring and restructuring. The system is continuously refactored to keep the design as simple, flexible and understandable as possible.

- Coding standards. The system development obeys the coding standards defined by the customer.

Several other XP practices are not used, because they are inappropriate for such a project.

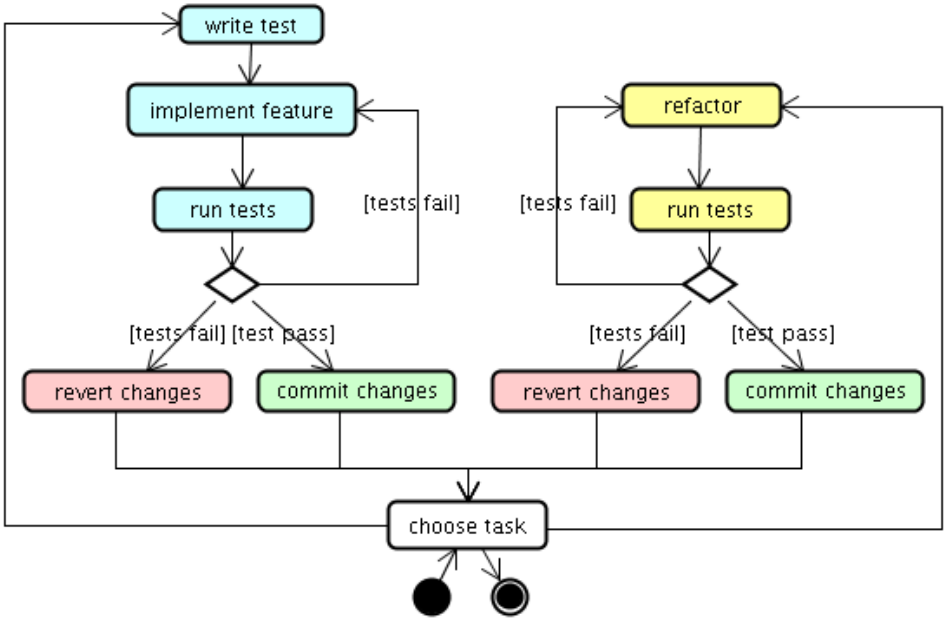

Figure 5.1: *Activities in the system implementation phase*

One important practice that was not used is test-first development on unit level. The internal structure of the software is expected to change massively as described in Section 5.2. Under such circumstances, the developer has made the experience that unit tests slow down the development because they have to be adapted, too. Therefore, unit tests are only written if critical, non-trivial units of the system that cannot be tested from specification level need to be tested. This is only possible because it is expected that the units of the software are not reused by other projects.

There are two basic activities (see Figure 5.1) in the system implementation phase:

- Feature implementation. Feature implementation begin with the creation of a test on specification level that passes if the software returns the expected result for the test metric specification executed on the test database configuration. Once the test creation is finished, the feature is implemented. After the implementation, all tests are executed. If they pass, the changes are committed to the version control system. If they fail, the developer decides if he tries to fix the implementation or reverts all changes since the last commit.

- Refactoring. The developer refactores the code and runs the tests. If they pass, he commits the changes into the version control system. If they fail, he decides if he tries to fix the implementation or reverts all changes since the last commit.

The process starts with choosing which task to do next and ends with it. Fixing software defects is handled in the same way as implementing a new feature: a test on specification level that fails because of the defect is written, and then the new "feature" is implemented, which means that the defect is fixed.

The tools used in the implementation phase are listed in Appendix A.

5.3.3 System Documentation and Rollout

In the system documentation and rollout phase, the system architecture was expected to be quite stable. It contains well chosen variation points that allow an easy accommodation of future changes.

The activities of this phase were:

- Documentation of the source code and architecture. The source code and the system architecture were documented according to the standards required by the customer. The documentation was done in such a late stage because it was expected that the software changes a lot in earlier phases, so a lot of documentation from earlier phases would be useless as it would be outdated. Because it was not needed for intra-team communication, the documentation was done in the last phase.
- User documentation. A user documentation was created. It explains how to use the software.
- Rollout. The software was installed on a server at Kisters AG.
- Minor changes and bugfixes. Minor changes requested by the users and fixes to software defects were implemented in this phase.
- Increasing the robustness of the software. Understandable responses to wrong user input were implemented.

Chapter 6

System Architecture

The chapter deals with the architecture and the design of the tool. It is divided in the following parts:

- General architecture. Description how the tool fits into the existing tool environment and in which modules it is split.
- General core module design. Description of the core module algorithm and the core module variation points.
- Detailed core module design. The different parts of the core module are explained in detail.
- General description of the chart and web frontend modules.

Most of the following sections contain a short problem description, a design description and an explanation of the design decisions. The design decision subsections explain the use of software architecture design principles [LL06, Chapter 17.3] like modularization, information hiding and separation of concerns in the discussed part of the system.

Several design patterns [GHJV94] are used in the system design. Examples are uses of the factory, facade, composite, adapter, null object and strategy pattern. In most cases, the design patterns are not explicitly mentioned here, but are visible in the class names.

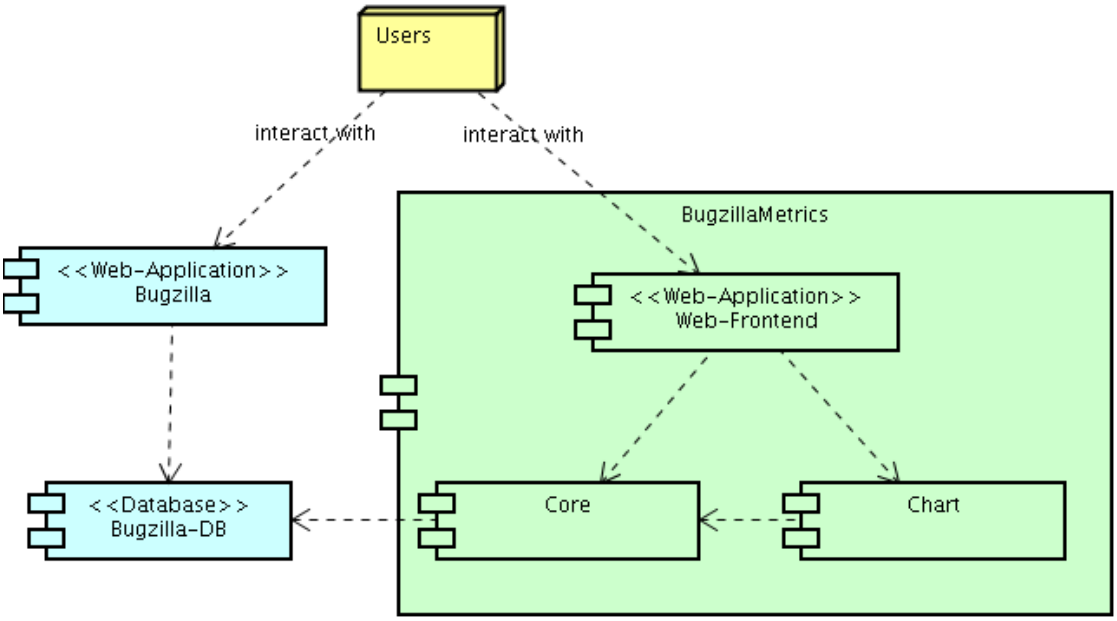

Figure 6.1: *Integration of the solution in the existing environment*

6.1 General Architecture

Problem Description

The tool must be integrated in the existing tool environment at Kisters AG. The tool should be split in different parts that can be reused by other tools.

Design Description

The tool was implemented in three different modules (see Figure 6.1):

- Core module. The core module has two XML interfaces, one for the metric specification and one for the metric result. It calculates the metric result for a metric specification.
- Chart module. The chart module is based on JFreeChart and has an XML interface for the chart configuration. It returns a `JFreeChart` object as result.
- Web frontend module. The web frontend module provides an HTML form through which the metric and chart configuration can be supplied, and calls the core and chart module to create the result that is displayed to the user.

The tool is integrated in the existing environment at Kisters AG as follows:

- The users use an existing Bugzilla installation to manage their cases.
- Bugzilla stores the cases in the Bugzilla-DB.
- The core module uses this database to retrieve the values for the cases. The database is accessed for reading purposes only.
- The users access the web frontend of the tool to get the results for their metrics.

Design Reasons

Their are two main reasons for the design that was chosen.

The first one is the requirement that the users should be able to use the tool, even if the user interface is very rudimentary. The second reason is reuse of the metrics component by other services.

Therefore, the following design decision were made:

- Use of a web frontend as user interface. This has the following advantages compared to a client application:
  - Lower maintenance cost. The web application has to be installed on a server only, each client can access it through a web browser. In contrast to that, a client application has to be installed on each client.
  - Use by other services. The interface of the web application is based on HTTP. Therefore, other services can call it and request chart images and metric result XML for further processing.
- XML interfaces. By using XML interfaces, services written in other languages, for example in Perl, can access the web frontend and use the metrics component. Furthermore, a future migration to web services is easier because they can be implemented as simple wrappers for the existing modules.
- Modularization into web frontend, core and chart module. By using this modularization, it will be easier to use the core and chart module independent of each other and the web frontend after converting them to web services.
- No modification of the existing environment. The existing, customized Bugzilla installation at Kisters AG does not need to be modified.

6.2 Core Algorithm Sequence

The basic interface to the algorithm is the class `CoreFacade`. It contains the method `doCalculate` that is parametrized by a string containing the metric specification in XML and writes the metric result as XML to an output stream.

The sequence of steps is the following (see Figure 6.2):

1. The `CoreFacade` parses the XML metric specification and configures the object structure of the metric calculation. The `MetricCalculation` object is the root of that structure. This step includes the configuration of the case value calculators, the reading of the group parameters and so on.

2. The core facade calls the `calculate` method on the configured `MetricCalculation`.

3. The `MetricCalculation` calculates which information is required by its configuration.

4. The `MetricCalculation` creates the cases required for the evaluation of the metric.

5. The `MetricCalculation` initializes the cases with the current values for the required fields.

6. The `MetricCalculation` calculates all case values by processing the event sources (see Section 6.5) and calling the configured case value calculators (see Section 6.7) with the created events. The case values are classified as they are stored in the case value container tree (see Section 6.8).

7. The `MetricCalculation` calculates the group values for the case values created in the previous step by calling the group value calculators with the bottom layer of the case value container tree (see Section 6.9).

8. The `MetricCalculation` creates the XML result element for the group values that are stored in the case value container tree.

9. The `CoreFacade` adds some status information like the metric configuration, a timestamp and the processing time to the resulting XML document and writes it to an output stream.

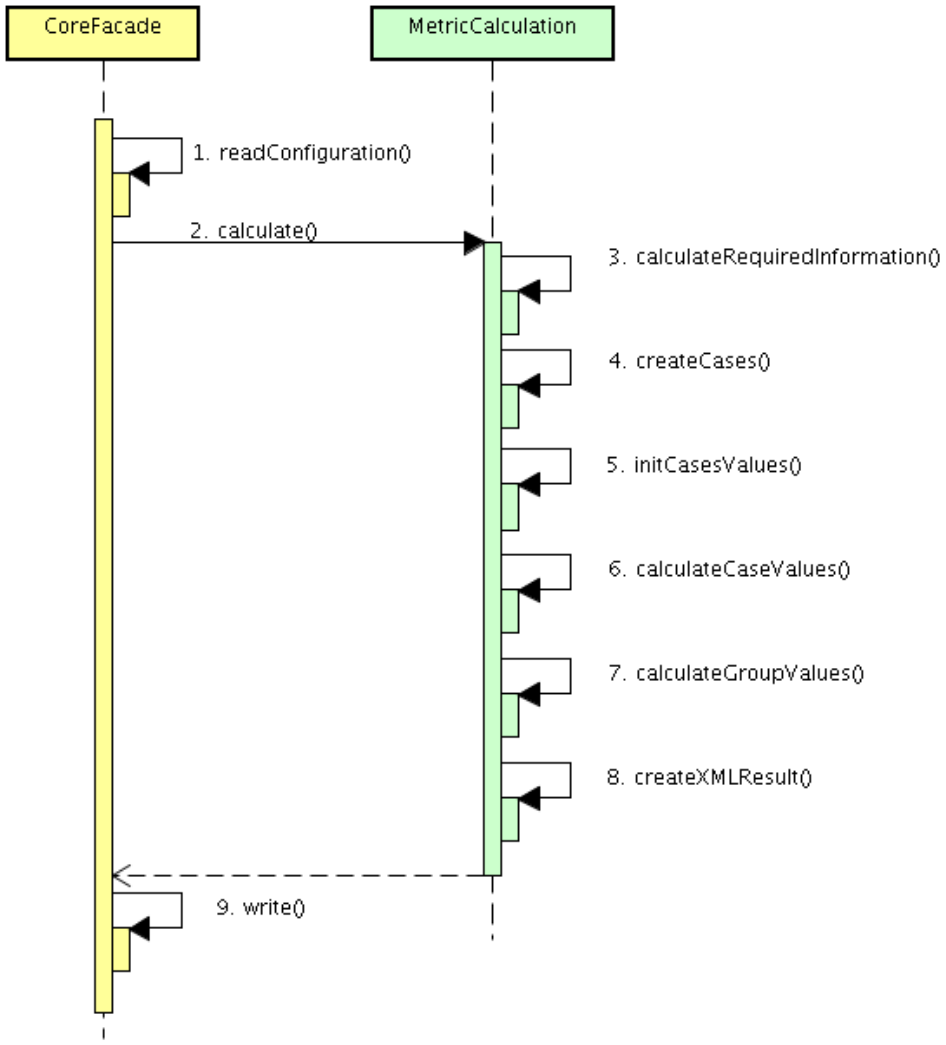

Figure 6.2: *General algorithm sequence*

Design Reasons

There are different reasons for the different aspects of the sequence.

The separation between the mapping of the metric specification to an object structure (`readConfiguration` method), the calculation itself (`calculation` method) and the serialization of the DOM to an output stream (`write` method) is a separation of concerns. The serialization and the metric specification mapping is not mixed with the algorithm itself, and they are independent of each other. As a result, the metric specification format and the output format can be changed without affecting the algorithm.

The algorithm itself is split into several steps. Each step does not know how the other steps work, they only communicate through the resulting data structures. Thus, the information how each step works and the internal data structures used in each step are hidden from the other steps.

The first three steps are the result of the data source separation and the optimization to calculate the required fields only (see Sections 6.11 and 6.13).

The case value calculation is separated from the group value calculation, because the different group value calculators should be able to use the results from the different case value calculators, and the complete case value calculation should be executed in one run over the case histories.

6.3 Core Algorithm Variation Points

Several points where the algorithm is likely to be extended were identified during the requirements engineering and the iterative development of the tool. These areas are examined in the following, grouped by their likelihood to change.

The following areas are very likely to change:

- Weights (see Section 6.7). Weights are used in the calculation of case values on certain events. New weights are likely to be added when new calculations for case values are required.
- Fields (see Section 6.10). Because the Bugzilla installation at Kisters is migrated to new versions and new features are added, it is very likely that fields are added or they need to be implemented in a different way, because their representation in the database changes.

- Data sources (see Section 6.11). For the same reason as for the fields, the data sources that provide the fields are likely to change.

The following areas are somewhat likely to change:

- Case state filters (see Section 6.4). New case state filters, for example to support range checks on number or date fields, might be added.
- Events (see Section 6.5). New event types might be added when new data sources are added.
- Event filters (see Section 6.6). New event filters that work on new event types or filter events differently might be added.
- Group calculation operations (see Section 6.9). New operations like more complex statistical or mathematical operations might be added to support new metric calculations.

The following areas might change, although they are expected to be stable after some changes:

- Case value calculators (see Section 6.7). New case value calculators might be added to support calculations that cannot be executed with the current calculators.
- Group evaluations (see Section 6.9). Group evaluations that create XML elements with information that differs from the existing group evaluations might be added. Currently, a details group evaluation and a calculating group evaluation exist.
- Groupings (see Section 6.8). The way the results are grouped might be changed. This includes changing the order and the available group parameters.
- Time granularities. Additional time granularities like quarter, year or hour might be added.
- Time period selection. Other mechanisms for time period selection, for example relative time periods with respect to the current date, might be added.

The tool is designed in a way that allows all these variation points to change by adding new classes, removing old classes and modifying existing implementations. The core algorithm of the tool and the basic data structures should not be affected by such changes, because the complete configuration is done by the configuration part and the core algorithm only knows the interfaces of the variation points, not the implementations.

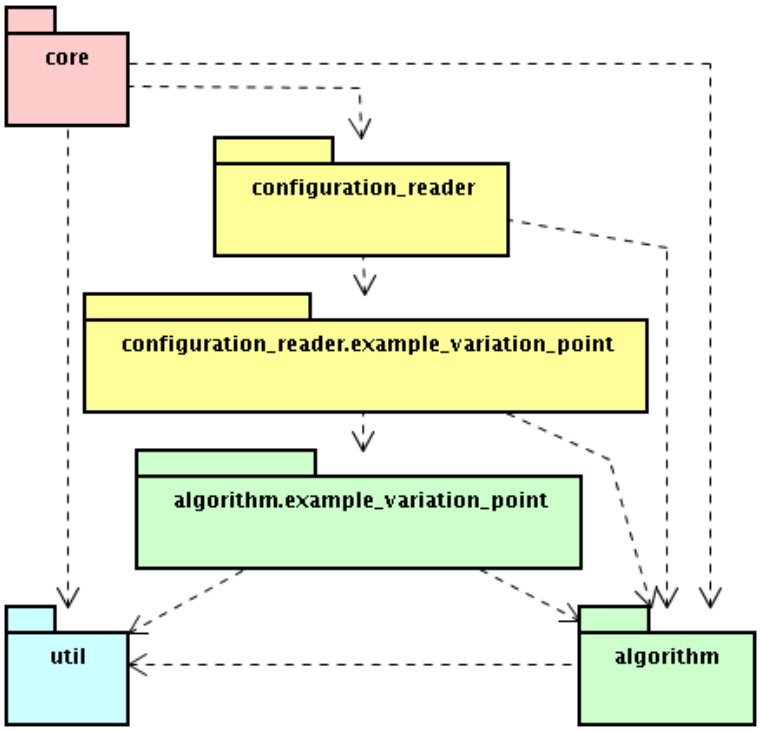

Figure 6.3: *Abstract design of the package dependencies*

Package Dependencies of Variation Points

The implementation classes for the variation points are localized in two packages, the package that contains the classes that implement the variation point and the package that contains the classes for the variation point configuration. The dependencies are as follows (see Figure 6.3):

- The core package which contains a facade accesses the configuration reader package to build the configured algorithm. Later, it accesses the algorithm package for the execution of the configured algorithm.
- The configuration reader package accesses the algorithm package because it creates and configures the algorithm. It also accesses the configuration packages for the algorithm variation points.
- The variation point configuration packages access the algorithm package because they configure some variation point of the algorithm. They also access the the variation point implementation package. Thus, the access to the variation point implementation package is restricted to the corresponding variation point configuration package.

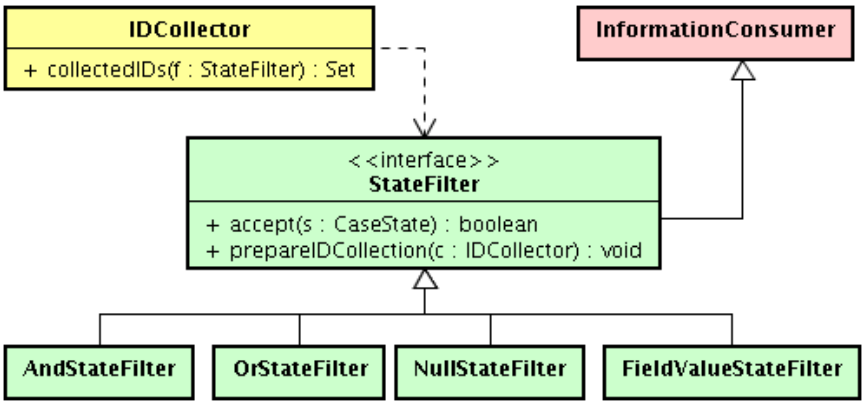

Figure 6.4: *Design of the case state filtering*

- The variation point implementation package accesses the algorithm package, because it implements some interfaces of the variation point that are defined in the algorithm package.
- Several packages use generic constructs supplied by utility packages.

This modularization and the structure of the dependencies have the advantages that common changes only have a local impact and that a clear separation of concerns is provided.

6.4 Case State Filtering

Problem Description

Cases must be filtered based on their current state when determining the base case set that should be evaluated and when filtering certain events. A flexible combination of the case state filters should be possible.

Design Description

A state filter (see Figure 6.4) is an information consumer (see Section 6.10). It provides an `accept` method that calculates whether a `CaseState` is accepted by the filter or not.

There are four different types of state filters:

- `FieldValueStateFilter`. This filter accepts all case states where the field value of the field checked by this filter matches the expected value of the filter.

- `NullStateFilter`. This filter implements the null object pattern and accepts all case states.

- `AndStateFilter`. This filter is a composite filter that combines its subfilters in a logical 'and' expression. To accept a case state, all its subfilters must accept it.

- `OrStateFilter`. This filter is a composite filter that combines its subfilters in a logical 'or' expression. To accept a case state, one of its subfilters must accept it.

Complex filters can be specified by the combination of `and` and `or` filters. For example, to filter a set of field values, the corresponding field value filters can be combined with an `OrStateFilter` filter.

The `IDCollector` has a `collectIDs` method that collects the identifiers of all cases that match the state filter passed as parameter. It is accessed for the calculation of the set of cases that will be evaluated by the algorithm. The collector calls the `prepareIDCollection` method on the state filters for the filter specific identifier collection.

Design Reasons

The clients, including the `IDCollector`, only use the `StateFilter` interface. Therefore, the only point where the subclasses are accessed is the creation of the filters.

This means that the state filter clients have no information about the state filters that are available or their implementation. Their responsibility in the interaction is just using the abstract state filter interface.

By separating the concerns in that way, the variation point that is likely to change, namely the addition and change of state filter implementations, is independent from the clients of state filters, and changes only have a local impact.

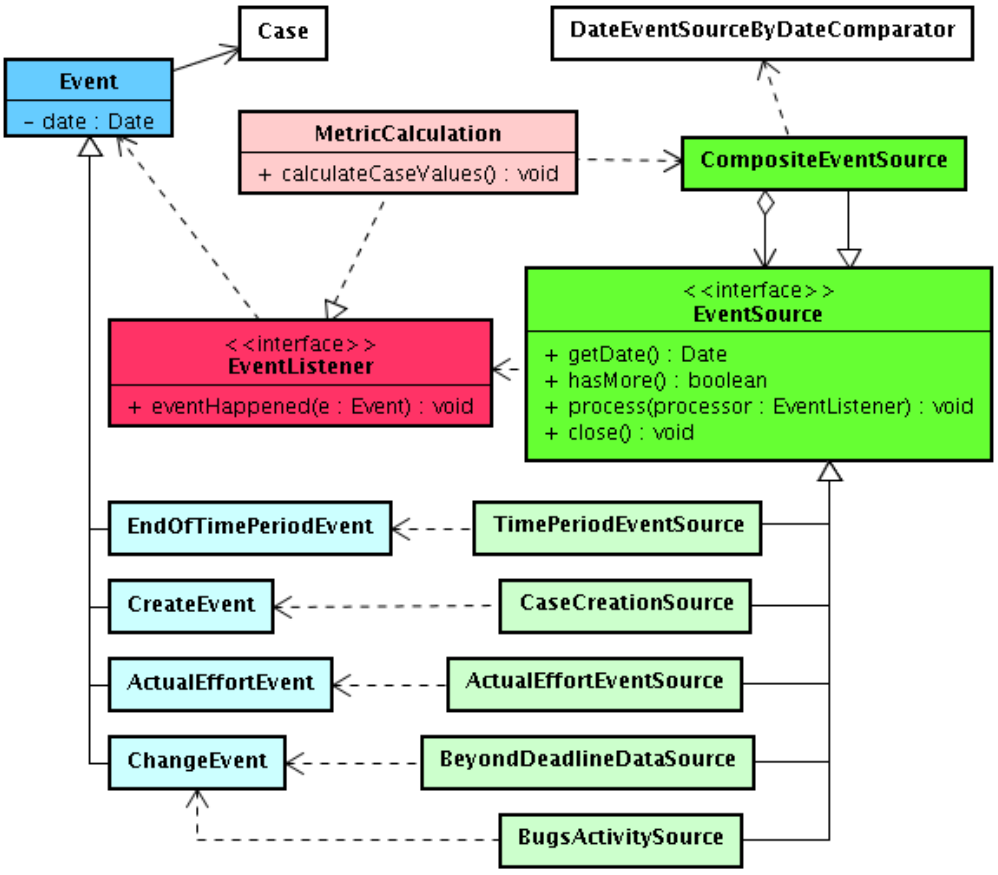

Figure 6.5: *Design of the event creation*

6.5 Event Creation

Problem Description

There are different types of events that should be processed by event consumers, e.g. case value calculators, in an inverted chronological order. The basic processing infrastructure should be extensible because new event sources and consumers might be added.

Design Description

An event source provides events in an inverted chronological order. Therefore, its interface contains methods to access the date of the next event, to check if there are any more events, and to process the next event. When

processing an event, an event listener is supplied that can be called if the source creates event objects. Finally, there is a method that is called to close a source after all events are processed or an error occurred.

The `CompositeEventSource` is an event source modeled after the composite pattern [GHJV94]. It unifies a set of other event sources and lets them appear as one. It uses a comparator that compares date event sources by their next date to provide the required inverted chronological order.

The following event sources are available:

- `TimePeriodEventSource`. This event source creates an event at the end of a time period.
- `CaseCreationSource`. This event source creates an event when a case is created and deletes that case from the list of available cases.
- `ActualEffortEventSource`. This event source creates an event when the actual effort changes.
- `BugsActivitySource`. This event source changes the values of case properties as certain fields change in the `bugs_activity` table and triggers change events.
- `BeyondDeadlineDataSource`. This event source changes the value of the boolean property "beyond deadline" to false once the current evaluation timepoint has moved back before the deadline of a case and triggers change events.

The interface `EventListener` has a method `eventHappened` that is called by concrete event sources when they create an event and want to notify the listener. It is implemented by the `MetricCalculation` class that forwards the events to the consumers.

The `MetricCalculation` class creates a composite event source from all sources and calls the method `process` in a loop as long as there are events left. It supplies itself as `EventListener` when calling the method `process`.

Design Reasons

In the original design, event creation and consumption was mixed in an algorithm that represented multiple sources and consumers in one class. The missing separation of concerns caused poor extendibility and maintainability.

In contrast to that, the advantages of the current design are:

- Good extendibility with respect to the event sources. New event sources can be added easily without modifying the basic infrastructure and the event consumers.
- Good extendibility with respect to the event consumers. New event consumers can be added easily without modifying the basic infrastructure and the event sources.
- Improved separation of concerns. Each event source just deals with its own events and changes, the sorting is done in the composite source and the event consumption is hidden through the `EventListener` interface.

6.6 Event Filters

Problem Description

The case value calculators (see Section 6.7) need to react on certain events and are therefore equipped with event filters. A flexible configuration of these event filters should be possible.

Design Description

An event filter (see Figure 6.6) is an `InformationConsumer` that has a method to filter events. The `accept` method returns a boolean that indicates whether the event passed as parameter is accepted by the filter or not.

The following implementations of the `EventFilter` interface are available:

- `BasicEventFilter`. This filter checks if the event is an instance of the `eventClass` of the filter. That way, it can be parametrized to filter any type of event, for example `EndOfTimePeriodEvents`.
- `CreateEventFilter`. This filter is a subclass of `BasicEventFilter` that accepts `CreateEvents`. It is an explicit subclass because it adds the requirement of including the creation timestamp field.
- `FieldChangeFilter`. This filter is a subclass of `BasicEventFilter` that accepts `ChangeEvents` and additional checks if the changed field is the field of the filter.

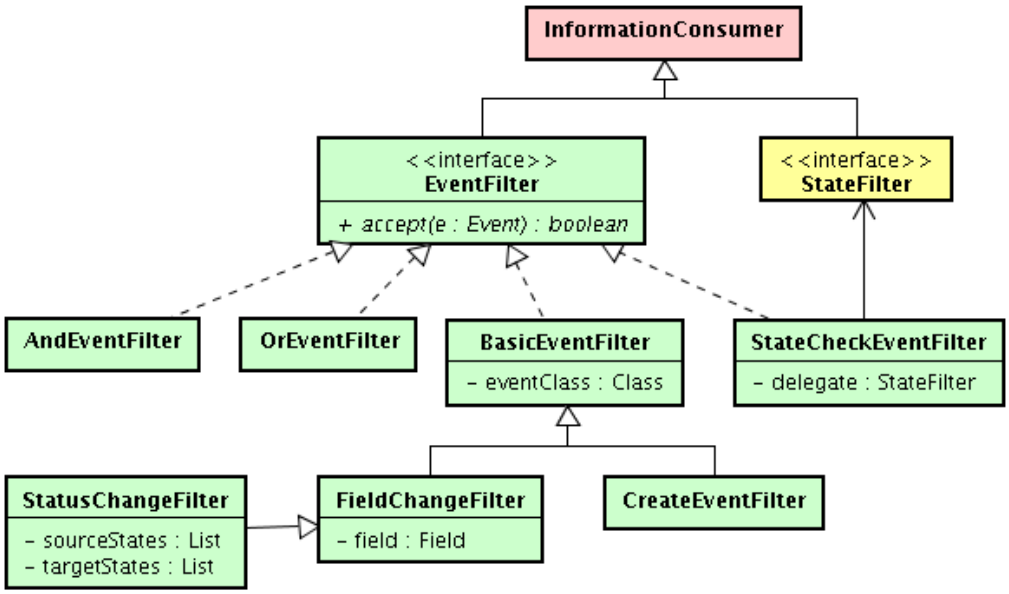

Figure 6.6: *Design of the event filters*

- `StatusChangeFilter`. This filter is a subclass of `FieldChangeFilter` that accepts `ChangeEvents` on the state field and additionally checks whether the state transition represented by the change event is a state transition from the filters' list of source states into its list of target states.

- `StateCheckEventFilter`. This filter checks the state of an event by delegating this check to the state filter it is parametrized with.

- `AndEventFilter`. This filter combines several event filters and accepts an event if it is accepted by all of the subfilters.

- `OrEventFilter`. This filter combines several event filters and accepts an event if it is accepted by one of the subfilters.

Design Reasons

The design reasons for the event filter design are somewhat similar to the reasons for the case state filter design. The clients only use the `EventFilter` interface. Therefore, the only point where the subclasses are accessed is the creation of the filters.

This means that the event filter clients have no information about the event filters that are available or their implementation. Their responsibility in the interaction is just using the abstract event filter interface.

By separating the concerns in that way, the parts that are likely to change in that variation point, namely the addition and modification of event filter implementations, are independent from the clients of event filters.

Additionally, by delegating to state filters in a special event filter, the potentials of the state filters are reused without any influences on the other event filters except the `StateCheckEventFilter`.

6.7 Case Value Calculation

Problem Description

The case value calculation with different case value calculators should create case value objects that are further evaluated by group evaluations. The whole calculation for all case value calculators should be executed in one run over the case history. The case value calculators should be configurable in a flexible manner, because their results are the base for a range of different metrics.

Design Description

The case value calculation (see Figure 6.7) consumes events created in the event sources, calculates case values based on these events and stores them in a case value container. For this purpose, a case value calculator contains a `calculateCaseValue` method.

There are three different types of case value calculators:

- `CountEventsCalculator`. This calculator counts events that match its filter, once an event has passed its `until` filter for that case.
- `IntervalLengthCalculator`. This calculator counts the length of an interval between two events for a case.
- `WeightCalculator`. This calculator is the most flexible calculator. When an event occurs that matches its filter, it calculates a value for that case based on its current state by delegating the main calculation to its weight.

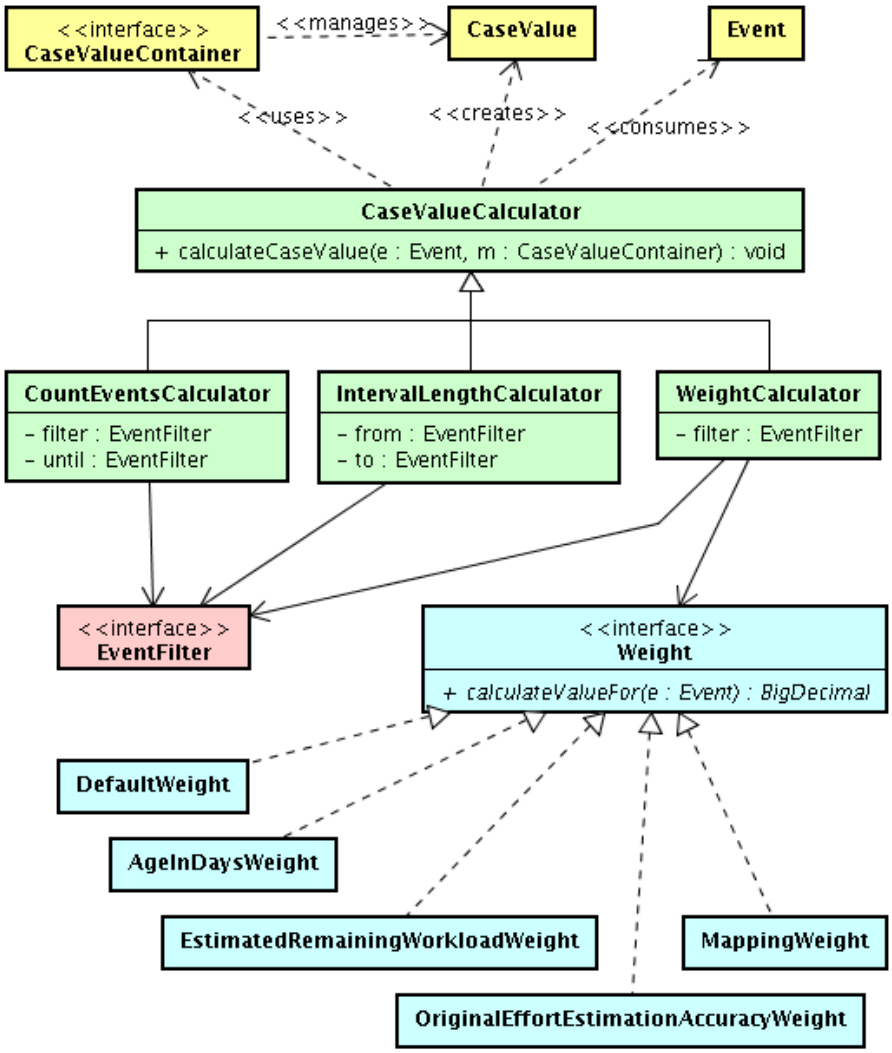

Figure 6.7: *Design of the case value calculation*

A `WeightCalculator` can be parametrized with one of the following weights:

- `DefaultWeight`. A weight that returns the number 'one' as value.
- `AgeInDaysWeight`. A weight that returns the age of a case on the occurrence of the event in days.
- `EstimatedRemainingWorkloadWeight`. A weight that calculates the estimated remaining workload in hours for that case on the occurrence of the event.
- `OriginalEffortEstimationAccuracyWeight`. A weight that calculates the accuracy of the original effort estimation by the following formula:

$$1 - \min(1, \frac{|\text{originalEstimatedEffort} - \text{actualEffort}|}{\text{originalEstimatedEffort}})$$

- `MappingWeight`. A weight that maps the value of a field to a number. It can be configured with a field and the mapping of field values to numbers.

Design Reasons

The three different case value calculators represent the three base cases of case value calculation that appeared when implementing the metrics from the requirements:

- Counting events
- Calculation of values on events
- Calculation of the time period length between two events

By using the `CaseValueCalculator` interface, the design is extensible if other types of case value calculators are required for new metrics.

The abstraction of using event filters to configure the case value calculators has two advantages:

- The different concerns of event filtering and calculating case values are separated. This especially provides independent extendibility of the event filter variation point and the case value calculator variation point.

- Flexible configuration of the case value calculators with different event filters.

The same applies to the use of weights in the `WeightCalculator` . It allows a flexible configuration with different weights and is in fact a separation of concerns between the calling of a calculation of a certain event and the calculation itself. Because it is likely that new weights are added, the effort of introducing them is reduced by this design.

6.8 Case Value Classification

Problem Description

The event based case value calculation creates case values. These case values should be accessible by the group evaluations and should be grouped as required for the evaluation. Additionally, the case value groups should be written to the XML result in the correct structure afterwards. Furthermore, a nested grouping of case values with different group parameters should be possible.

Design Description

Case value calculators (see Figure 6.8) create case values and store them in case value containers. Because the case value containers form a tree, the case value calculators only access the root container. The correct place for storing a case value is found by recursion in the tree. The case value calculators use the methods `add` and `remove` to add and remove case values. If they need to remove case values later, they have to store references on the case values objects that might be removed by themselves, too.

A case value has the following attributes:

- Date. The date of the event for which the case value was created.
- Value. The value that the case value calculator calculated for that event.
- State. The state of the case when the event happened.
- Case value calculator identifier. The identifier of the case value calculator that created the case value.
- Case identifier. The identifier of the case that the event is related to.

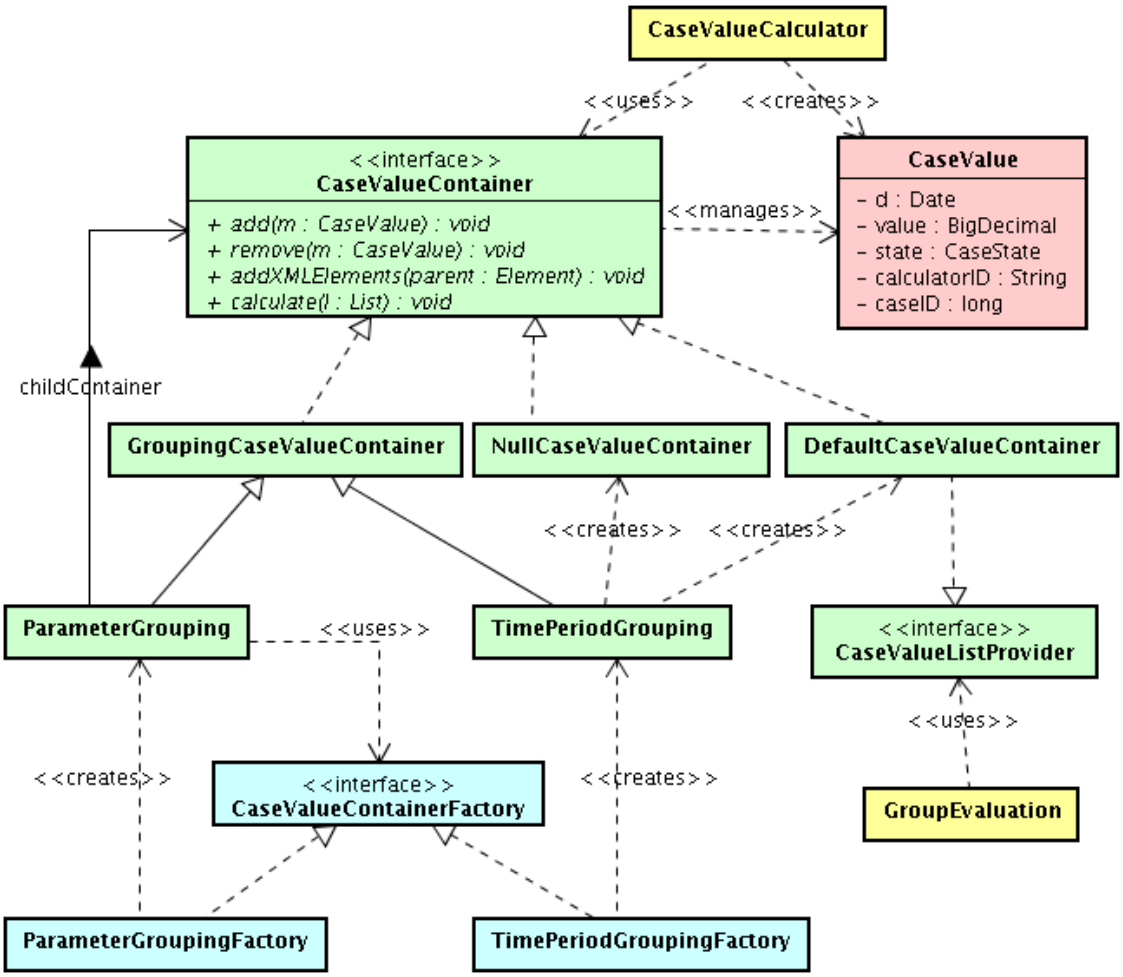

Figure 6.8: *Design of the case value classification*

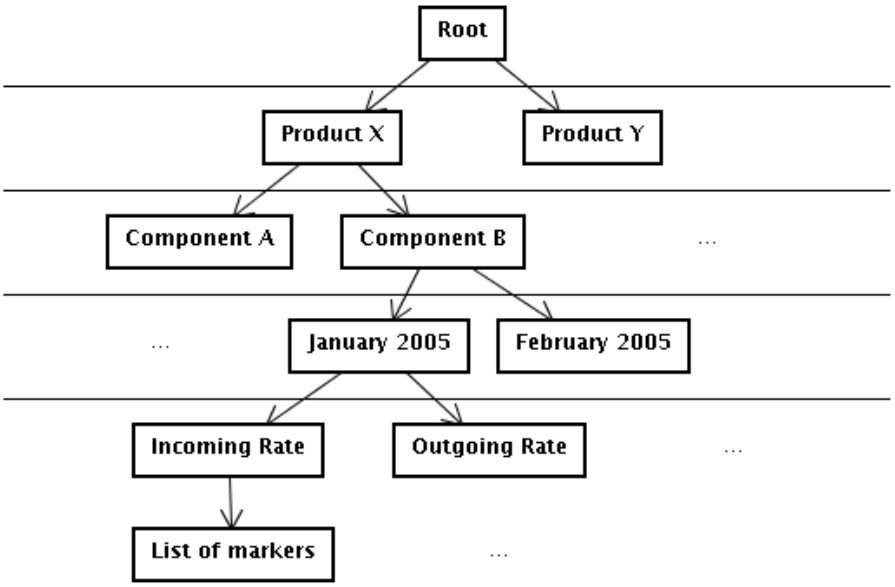

Figure 6.9: *Example for a case value container tree*

Based on these attributes, the classification tree is split into the following different layers (see Figure 6.9):

- Parameter based grouping. There can be one or more tree layers that group the values according to a certain group parameter. Based on the value of the case state for the field specified by this parameter, the case value is forwarded to the correct child container.
- Time period based grouping. Beneath the layers of parameter groupings, the case values are groupings according to the time period they fall into. If they are outside the valid range of time periods, they are added to a `NullCaseValueContainer`, which is an implementation of the null object pattern.
- Case calculator based grouping. In this layer, the case values are grouped based on their case value calculator identifier.

In each layer of the tree, different types of case value containers are used.

In the parameter based grouping layer, `ParameterGroupings` are used. They use `CaseValueContainerFactories` to produce their child groupings, which can be `ParameterGroupings` or `TimePeriodGroupings`. Therefore there can be many layers of `ParameterGroupings`. They distribute case values according to the evaluation of the case state with respect to the group parameter.

`TimePeriodGroupings` are used in the next layer. They create `DefaultCaseValueContainers` for all valid time periods and use a `NullCaseValueContainer` for case values that are outside that range. They distribute the case values to the correct child container with respect to the time period they fall in.

`DefaultCaseValueContainers` are used in the last layer. They distribute case values according to the case value calculator that created them.

The group evaluation is started by a call of `calculate` on the root case value container with a list of the group evaluations as parameter. The method recurses through the tree to all `DefaultCaseValueContainers`. These containers call the `createResultElement` method of the group evaluation with themselves as parameter. This is possible because they implement the `CaseValueListProvider` interface required by the group evaluations. The resulting XML elements are stored in a list in the `DefaultCaseValueContainers`.

The XML result creation is also run over the structure of the tree. On the root case value container, the `addXMLElements` method is called with a root element as parameter. Each layer adds its own element that is filled with the elements of its children by recursion. In the `DefaultCaseValueContainers`, the results from the group evaluation are added. This way, the XML result tree is created.

Design Reasons

The main reasons behind this design are modularization and separation of concerns. The concern of the case value containers is to classify the case values and to create the structure of the XML result.

The case value calculators, which create the case values, only know the case value container interface and are unaware of the composition of the tree. They only access the root of the tree.

The group evaluations, which calculate XML elements for sets of case values, are only called with objects that implement the `CaseValueListProvider` interface required by them, and do not know the rest of the structure. They work at the bottom of the tree.

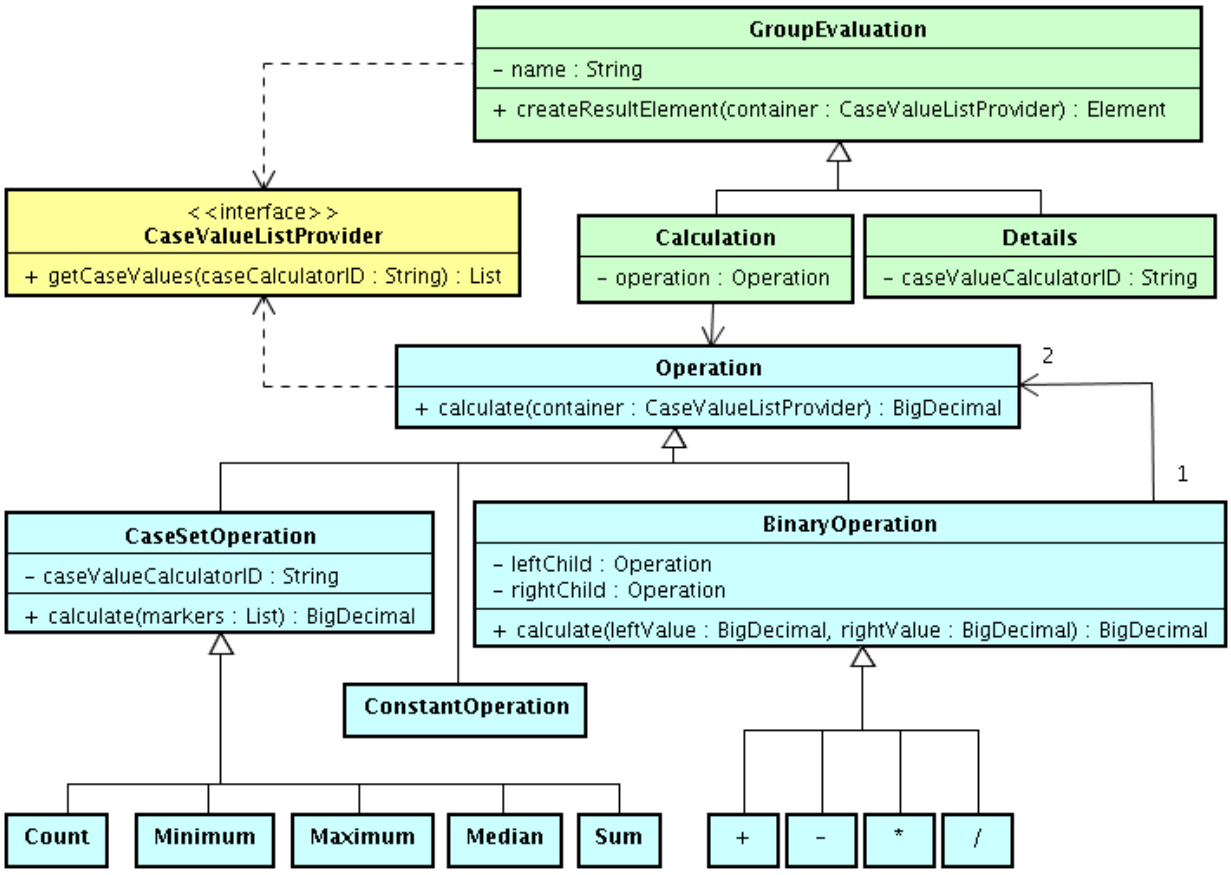

Figure 6.10: *Design of the group evaluation part*

Therefore, the complete classification and tree composition and therefore result composition can be changed without affecting the case value creation and the group evaluation.

Another important reason is that the tree structure reflects the result structure to simplify the mapping to XML. Each layer of the tree is mapped to a layer of XML elements.

6.9 Group Evaluation

Group evaluations (see Figure 6.10) create an XML element for the metric result DOM by evaluating a set of case values. The set of case values is restricted to a certain time period and case group, although this is unimpor-

```
<details name="outgoing rate">
  <case id="1" value="1" />
  <case id="2" value="4" />
</details>
```

Figure 6.11: *Detail evaluation result example*

tant for the group evaluations themselves. The case values themselves were calculated and grouped in the case value calculation phase.

Each group evaluation has a name which is used as name attribute for the resulting XML element. The name of the XML element itself is determined by the evaluation type. Its method `createResultElement` is called with different `CaseValueListProviders` for the different groups and time periods in the group value evaluation phase. It creates the XML element and fills it with content that differs with the subtypes of `GroupEvaluation`.

The method `getCaseValues` in the interface `CaseValueListProvider` returns the list of case values for the specified case value calculator. This list can then be evaluated further by the group evaluations.

There are two basic kinds of group evaluations, calculations and detail evaluations.

Detail Evaluation

The detail evaluation lists the case identifier and the value of each case value of the relevant case value calculator (see Figure 6.11). It is parametrized by the identifier of the case value calculator.

Calculations

Calculations are complex evaluations that calculate numbers for the case values passed in a `CaseValueContainer`. A calculation has a root operation on which it calls calculate with the `CaseValueContainer` to get the calculated `BigDecimal`. This value is used as text for the resulting XML element.

There are three basic kinds of operations:

- Case value set operations. Case value set operations calculate a value for a set of case values of the relevant case value calculator. They

```
<calculation name="backlog management index 2" >
  <divide>
    <subtract>
      <sum caseValueCalculator="outgoingRate" />
      <sum caseValueCalculator="incomingRate" />
    </subtract>
    <add>
      <sum caseValueCalculator="outgoingRate" />
      <sum caseValueCalculator="incomingRate" />
    </add>
  </divide>
</calculation>
```

Figure 6.12: *Calculation specification example*

are parametrized with the identifier of that case value calculator. The different types of case set operations are:

  - Count. Counts the number of case values in the set.
  - Minimum. Calculates the smallest value of the case values in the set.
  - Maximum. Calculates the largest value of the case values in the set.
  - Median. Sorts the case values by their values and returns the value of the case value in the middle.
  - Sum. Sums the values of the case values.

- Binary operations. Binary operations call two child operations and combine their results in a specific way to one result. The different types of binary operations are:
  - Addition. Adds the results of the child operations.
  - Subtraction. Subtracts the result of the right child operations from the result of the left one.
  - Multiplication. Multiplies the results of the child operations.
  - Division. Divides the result of the left child operations by the result of the right one.
- Constant operation. Returns a value specified in the metric specification.

The operations form a tree with the binary operations as inner nodes and constant and case set operations as outer nodes. This tree is evaluated recursively. An example of a calculation specification is given in Figure 6.12.

Design Reasons

The main reasons for designing the group evaluation in such a configureable way are flexibility and reuse.

In the context of group evaluations, reuse means that the evaluations of different metrics were split into their basic parts, which are operations like sum or divide. These operations are reused in different metrics now. Once an operation is available, it can be used in any group value calculation.

Flexibility means that by composing the basic parts, new calculations for new metrics can be defined in the request specification without changing the software, assuming they compose the available operations in some new way and do not require additional operations.

These two properties result in a lower cost when calculating the values of new metrics, compared to a solution were the metrics are defined in a fixed way in the tool itself. Because it is unclear how the future use of the tool will evolve with respect to the metrics calculated, a scenario where new or modified metrics should be evaluated is likely.

6.10 Information Requirement Abstraction

Problem Description

A case consists of different fields like the product it belongs to, the creation timestamp and so on. In the modified Bugzilla installation used by Kisters, not all fields are stored in the same way. There are the following different kinds of storage for case fields:

- Columns in the bugs table. The original fields from Bugzilla are stored in the original Bugzilla way as columns in the bugs table.

- Rows in the `bugs_customfields` table. Some of the new fields that were added are stored as rows in the `bugs_customfields` table. Each row contains the case id, the property id and the value.

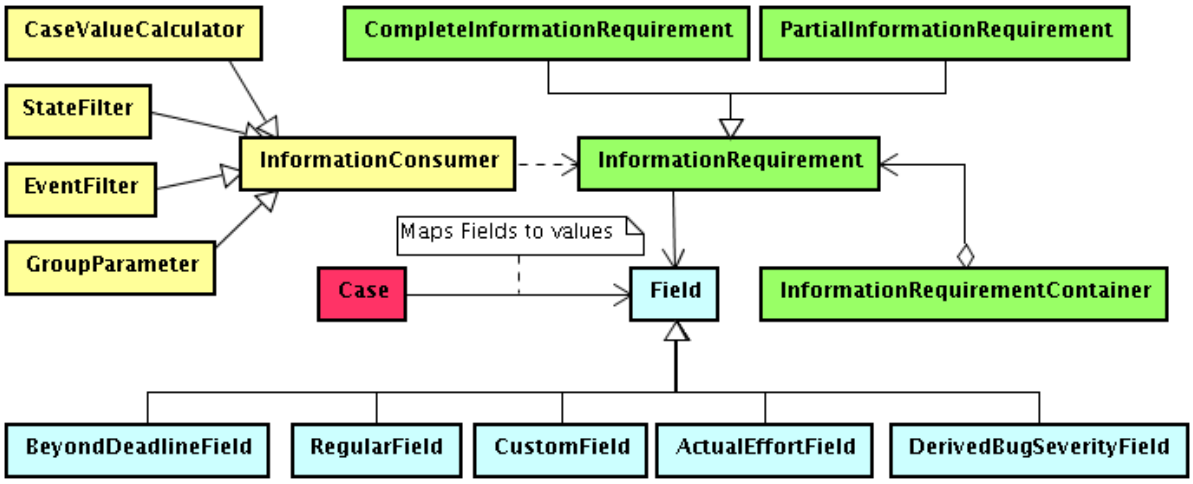

Figure 6.13: *Design of the information requirements*

- Reconstruction from the `bugs_activity` table. The actual effort field is reconstructed from the `bugs_activity` table by selecting all the rows with a given case id and the actual effort field id and adding the values from these rows.
- Splitting of fields from the bugs table. The type and the severity of a case are both encoded in the `bug_severity` column of the bugs table. To derive the type and the severity of a case, the value of these fields needs to be split.
- Derivation of values depending on the current history date and the value of another field. An example for that is the question if a case is beyond its deadline at given point of time or not. Such derived fields are important for flexible filtering, for example.

The usage of the case fields should be independent from their storage and implementation.

Design Description

`InformationConsumers` (see Figure 6.13) are objects that require information about certain case fields. The following classes are `InformationConsumers`:

- `CaseValueCalculators`. They need information about the fields based on which they calculate values. Furthermore, they need the information required by their event filters.

- `StateFilters`. They require information about the fields their case state filtering is based on.
- `EventFilters`. They may include state filters and require the information required by the state filters.
- `GroupParameters`. They require information about the field for which the grouping splits the case values into different groups.

`InformationConsumers` have `InformationRequirements`. There are two types of `InformationRequirements`: complete and partial information requirements.

- `PartialInformationRequirements` are information about certain values of a given field. For example, for a state filter that filters on a certain product, not all information about product values in cases is necessary, only information that affects that product.
- `CompleteInformationRequirements` return all information about a certain case field. This is the case for group parameters, which do not know which groups there are in advance.

`InformationRequirements` are collected in an `InformationRequirementContainer`. This container automatically consolidates the requirements by unifying `PartialInformationRequirements` and replacing `PartialInformationRequirements` with `CompleteInformationRequirements` when appropriate.

All `InformationRequirements` are directed on one specific field, which is one case field. For example, a group parameter that creates the groups according to the assignee has a `CompleteInformationRequirement` about the field assignee, which is a regular field stored in the bugs table.

For the different types of case field representations in the database, there are different types of fields:

- Regular fields represent case fields that are stored as columns in the bugs table.
- Custom fields represent case fields that are stored as rows in the `bugs_customfields` table.
- The actual effort fields represents the actual effort value reconstructed from the `bugs_activity` table.

- Derived bug severity fields represent the fields derived from the bug severity regular field. These are the type and the severity field.

- The derived beyond deadline value is represented by the special `BeyondDeadlineField`.

A case maps fields to the current values of these fields. It has no information about the way they are stored.

Design Reasons

The goal of this design is to achieve independence of the use of the case properties from the way they are stored and retrieved. These different concerns are separated by this design.

By hiding the details of the storage of the case fields, the following advantages are achieved compared to an implementation where those details are not hidden:

- Reduced complexity for using the case fields. When using case fields in the information consumers, the software developer does not need to think about the way these fields are stored.

- Independent extendibility. The storage of the case fields can change without affecting the `InformationConsumers` and the `InformationConsumers` can change without affecting the field representation. This especially means that both can be extended and modified without affecting each other.

- Additional performance optimization at a higher level. The `InformationRequirementContainer` can optimize the `InformationRequirements` by unifying and replacing them before they are passed to a lower level. This makes the optimization on the lower levels simpler. Additionally, the optimization on the higher level is more robust assuming the lower levels are more likely to change.

The main disadvantage is the increased design complexity introduced to achieve the separation of concerns.

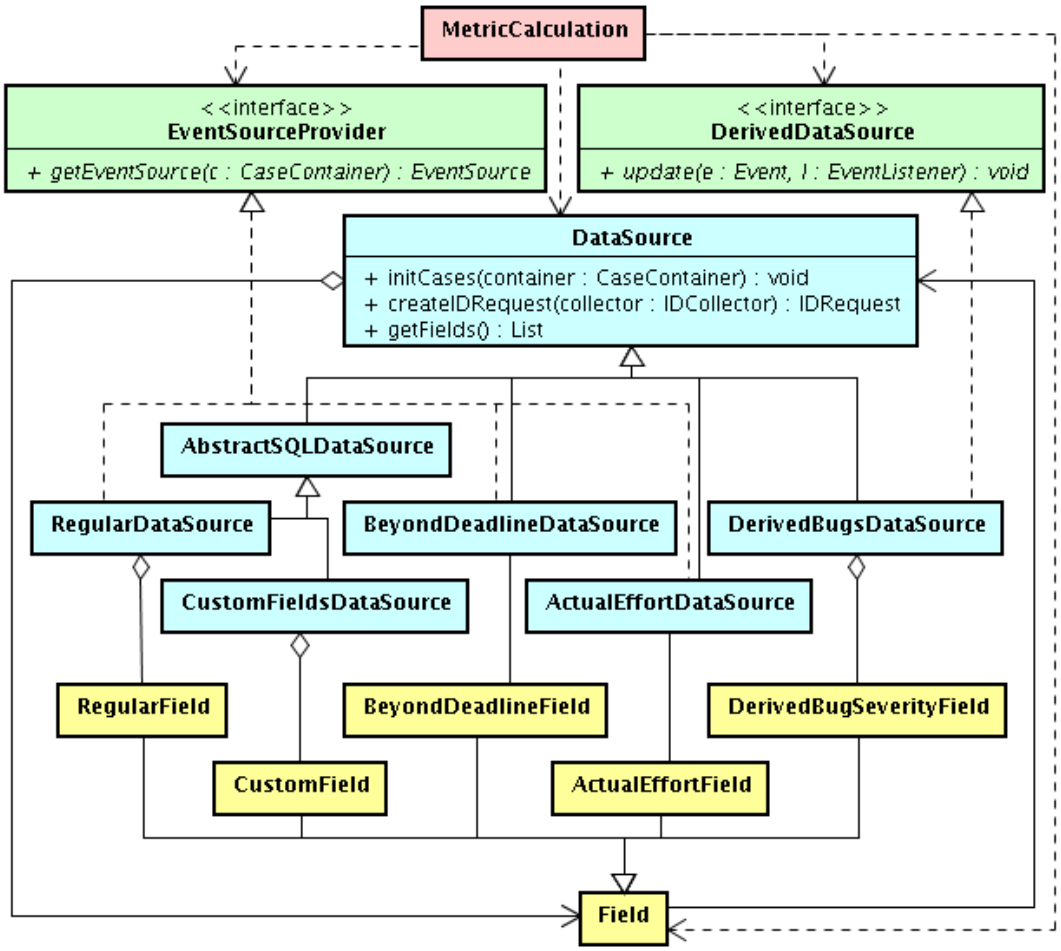

Figure 6.14: *Design of the data source abstraction*

6.11 Data Source Abstraction

Problem Description

The different field types are implemented in different ways. But their implementation is not the only difference, searching and accessing them is different, too. Furthermore, they have different associated event sources.

Design Description

For each of the different field types, a special `DataSource` exists (see Figure 6.14). Each data source provides a specific implementation of the following features:

- Fields. Each data source returns the list of fields it provides. The fields are usually objects of a data source specific implementation of the `Field` class (see Section 6.10). The fields are accessed through the `getFields` method.
- Search functionality. Each data source provides support for searching cases by values of their fields. This search functionality is implemented in a data source specific way. This is done by implementing the `createIDRequest` method.
- Field initialization. Each data source provides support for the initialization of the current values of its fields in a case. The fields are initialized in the `initCases` method.

Optionally, a data source can implement the following interfaces:

- `EventSourceProvider`. Data sources implementing this interface provide an `EventSource` that is used in the event creation. The data source specific implementation of the event source must take care of updating the relevant cases on events.
- `DerivedDataSource`. Data sources that depend on other data sources because their fields are derived from some other fields implement this interface. `DerivedDataSources` are called upon change events and can update the derived fields in cases. After updating the values, they should send events regarding their changes to the `EventListener` that is given as parameter.

The `MetricCalculation` object depends on the abstract `DataSource` class, the `Field` class and the additional interfaces. It does not know the concrete data source implementations. In the configuration phase, the data sources are created and the metric calculation is parametrized with them.

Design Reasons

The main design reasons for this kind of abstraction where already discussed in the design reasons subsection of Section 6.10.

6.12 SQL Access Abstraction

Problem Description

Different data sources access the SQL database of Bugzilla with different SQL queries that are calculated based on the information requirements of

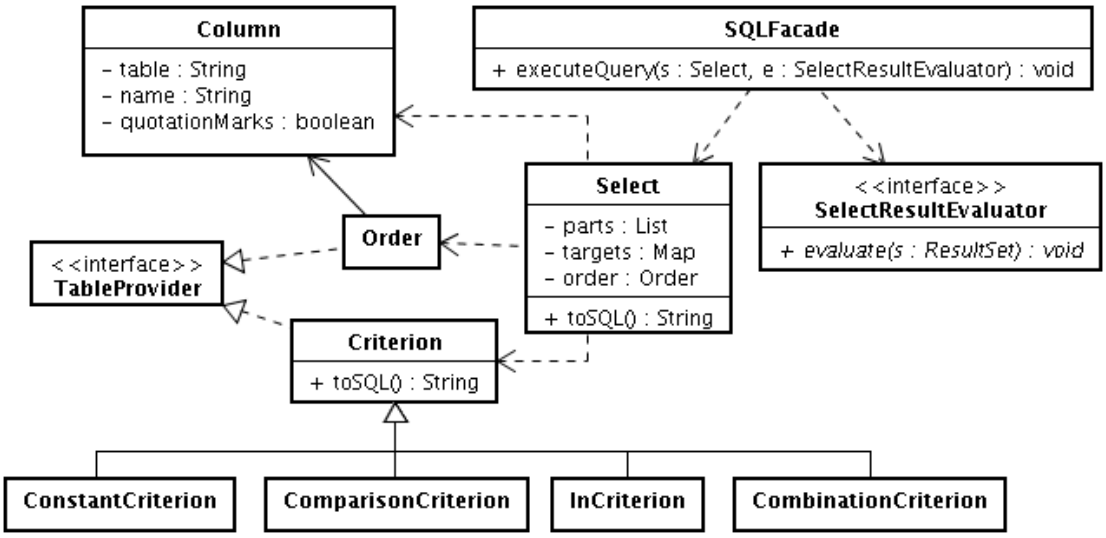

Figure 6.15: *Design of the SQL abstraction layer*

the current metric. The SQL queries should optimize themselves. Only a sub-part of the SQL query language is used.

Design Description

The method `executeQuery` of the `SQLFacade` (see Figure 6.15) encapsulates the access to the SQL database. It executes select queries, which are supplied as an `Select` object parameter. For the result evaluation, `executeQuery` calls the callback interface `SelectResultEvaluator` for each row in the result set. Thus, it hides a large part of the complexity of using the `java.sql` API from the client.

The `Select` class represents a SQL select query. It contains different parts which are `Criteria`, and a list of target `Columns` for each part. The SQL select strings generated for each part are joined by SQL union statements and the order is specified.

A column has a name, a table it belongs to and a flag whether its type needs quotation marks or not. For example, integer columns do not need quotation marks, whereas timestamp columns do. This was the simplest sufficient way to model SQL columns for the select queries required by the clients.

The `Order` class has a column that the order is computed for and a flag whether the order is ascending or descending.

The `Criterion` class models the `WHERE` part of the SQL statement. Criteria can form a tree and are evaluated recursively in the `toSQL` method that converts the criterion to the `WHERE` part of the SQL statement.

The `Criterion` class acts as facade for creating certain criteria, too. The subclasses except `CombinationCriterion` are only visible in the sql package. The following methods are provided by the `Criterion` class for creating criteria:

- `And`. Creates a `CombinationCriterion` that serves as a SQL `AND` statement.
- `Equals`. Creates a `ComparisonCriterion` that serves as a SQL `=` statement.
- `Everything`. Creates a `ConstantCriterion` that serves as a SQL `TRUE` statement.
- `GreaterThan`. Creates a `ComparisonCriterion` that serves as a SQL `=>` statement.
- `In`. Creates a `InCriterion` that serves as a SQL `IN` statement. An `InCriterion` optimizes itself to a `=` or `TRUE` statement for the special cases of a set with one respectively zero elements.
- `Or`. Creates a `CombinationCriterion` that serves as a SQL `OR` statement.

Both criteria and order objects are `TableProviders`. The `Select` class uses table providers to calculate the `FROM` part of the select query.

Design Reasons

The SQL statements are calculated based on information requirements that vary from metric to metric. By identifying the abstract elements that vary and using them as input for the creation of the SQL query string, the concerns of specifying the query and translating it to valid SQL are separated.

The clients only need to specify on a high level which rows should be selected, which columns should be selected and the way the result should be

ordered. Because the specification is done in an object structure, one query can be specified by the cooperation of several clients that do not know each other.

Furthermore, by hiding the creation of the SQL statement itself, the SQL abstraction layer can optimize the generated SQL statement. The `InCriterion` and the `CombinationCriterion` do that by handling special cases differently. The clients remain simpler because they do not contain the SQL statement generation and optimization.

By encapsulating the SQL statement generation and optimization in one package, the logic can be reused from different clients that access the SQL database.

6.13 Efficiency Optimizations

Execution Time Optimizations

The algorithm itself was designed considering the overall execution time. The following design decisions were taken in advance on algorithm design level with respect to the execution time:

- The set of cases that are evaluated is restricted by a base filter.
- The case value calculation phase is only executed once to reduce the amount of queries that retrieve values from the database.
- All case value calculators calculate their values during the same single run over the history of the cases.

Further optimizations were implemented after the algorithm was tested with real data, because it did not terminate within several minutes on a test system:

- Unnecessary join statements between the `bugs` and the `bugs_activity` table were removed. This improved the execution time of database queries significantly.
- SQL `IN` statements were used instead of `OR` statements for the selection of cases by their identifiers from the `bugs`, `bugs_activity` and `bugs_customfields` tables. This further improved the execution time of those queries significantly.

- If possible, the time period for which the history is reconstructed is limited to the evaluation time period specified in the metric specification. This means that the SQL queries on the `bugs_activity` table contain that time restriction and return a limited set of results only.
- The `Criteria` from the object structure that represents an SQL `SELECT` query optimize themselves when they are converted to SQL:
  - `IN` converts to `FALSE` for the special case of set size zero.
  - `IN` converts to `=` for the special case of set size one.
  - `AND` converts to `FALSE`, if at least one child of `AND` is `FALSE`.
  - `OR` converts to `TRUE`, if at least one child of `OR` is `TRUE`.
  - `AND` converts to the result of its child, if it has only one child.
  - `OR` converts to the result of its child, if it has only one child.
- It is checked whether the `WHERE` part of SQL statements is `FALSE` or not. The SQL statements are only executed if the `WHERE` part is not `FALSE`.
- `InformationRequirements` are automatically consolidated when added to an `InformationRequirementsContainer`. Such a container contains at most one requirement per field because of this consolidation.
- In the base case set selection, the `InformationRequirements` of `or` parts are consolidated in a container. Therefore, the number of SQL queries for `or` parts is reduced.

After implementing the optimizations mentioned above, the results for the different metrics where calculated in less than one minute on a test system with the real data. Most metrics were calculated in less than twenty seconds on that system. The fastest metrics were calculated in less than five seconds.

The following optimizations were planned at first, but they were not implemented due to different reasons:

- In the base case set selection with `and`, multiple SQL queries are executed. The idea was to add the set of possible valid identifiers as constraint to the SQL queries. Unfortunately, this was a lot slower than executing the set cut operation in the tool itself.
- In the base case set selection, similar criteria for `and` should be executed in one SQL statement. Unfortunately, this turned out to be problematic because there is a combinatorial explosion of possibilities.

Memory Consumption Optimizations

When testing the tool for the first time with real data, `OutOfMemoryErrors` occurred because there were problems with the memory usage efficiency. The following changes were implemented to cope with that:

- The creation of new immutable objects with the same state was replaced by using the same immutable object at the following occurrences:
  - When retrieving the current state of a case, a new case state object is only created if the case has changed. Otherwise, the current case state object is returned. This prevented an `OutOfMemoryError` that existed because new case state objects were created at the end of each time period in certain case value calculators configurations.
  - The `BigDecimal` objects with the values zero and one, which are used quite often in weights and calculations, are stored as constants in a special class called `Numbers`.
- A special hash map that uses linear probing and expansion by doubling [Sed03, Chapter 14] for `long` values and `Identifiable` objects was implemented. Both `Case` and `CaseValue` objects are `Identifiables`. This hash map does not need wrappers for the `long` values and therefore consumes less memory and is faster than the default implementation in the `java.util` package.
- The XML output is written to an output stream instead into a string. Therefore, the result string which can be very large is not stored in the memory.

After these optimizations, `OutOfMemory` errors disappeared and the execution time was further reduced, too.

6.14 Test Framework

Problem Description

The test driven development approach of the software process requires a testing framework that simplifies the implementation of tests on specification level. JUnit, the default approach for unit testing in Java, is not suited well for such tests, because it is primarily designed for tests on unit level.

Design Description

The tests are specified in XML files and plain text files. There are four different kinds of files, all in different directories:

- Expected result files. These files contain the expected metric result XML document. Their name is the name of the database configuration file and a dot followed by the name of the metric request file. By this configuration, they specify the expected result for running the metric request over the database configured with the database configuration.
- Metric request files. These files are referenced from the name of the expected result file and contain a metric configuration XML document.
- Database configuration files. These files are referenced from the name of the expected result file, too. They contain a database content setup specified in XML.
- Expected SQL query files. These files are optional. If for an expected result file an expected SQL file exists, the SQL queries are checked, too. The file contains an ordered list of SQL statement, one per line.

The test framework reads the estimated result files. For each expected result, it sets up the database according to the referenced database configuration file and executes the algorithm with the content of the referenced metric request file as input. The result is compared to the content of the expected result file. If an expected SQL query file exists, the send SQL statements are checked, too.

The test framework itself is implemented as a JUnit test case with several helper classes.

Design Reasons

This design has the following advantages compared to testing without such a framework, especially compared to testing on unit level:

- The test cases are on specification level.
  - The test cases are more stable against refactorings, because they are on a higher level than unit tests.
  - The test cases serve as a reference for the specification of metric requests.

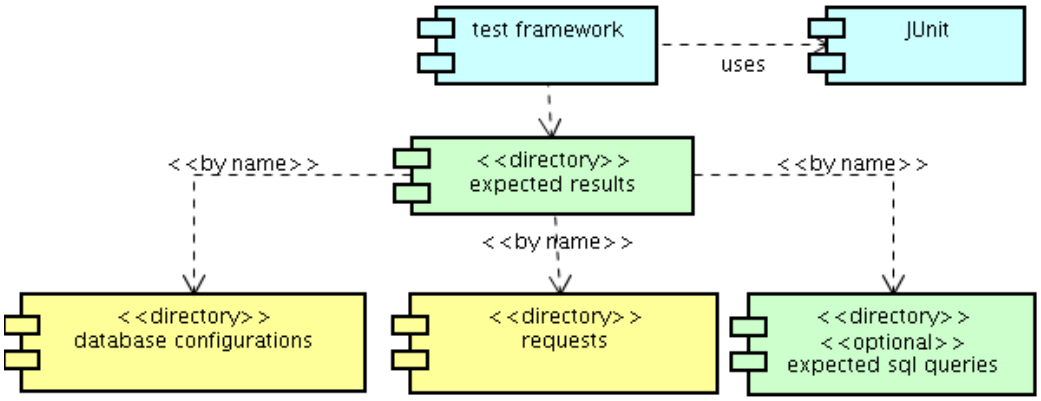

Figure 6.16: *Design of the test framework*

- The test cases are independent of the programming language.
  - For specifying test cases, programming knowledge is not required.
  - The test cases are portable to other platforms. For example, if the complete tool would be redeveloped in C#, the test cases could be reused with a different test framework. It could be checked that both the Java and the C# implementation of the tool would produce the same results on the same inputs.
- The test framework is rather small. With the DB setup classes, it consists of seven classes with 476 LOC. The test class itself has only 166 LOC.
- Database configurations and metric requests can be reused in different expected metric results.

Unfortunately, there are some disadvantages and limitations, too:

- The units of the tool are not tested directly. For that reason, reusing them independently is problematic.
- Some minor changes to the design where necessary to achieve the required level of testability:
  - Additional sorting in the algorithm was added to achieve a stable and therefore testable output.
  - A special test hook was added to the SQL access abstraction part for testing the SQL queries.

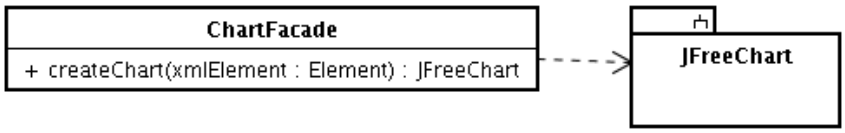

Figure 6.17: *Design of the chart module*

- When the metric request format changes, all metric requests must be adjusted.
- When the metric result format changes, all expected metric result files must be adjusted.

6.15 Chart Module

Problem Description

The chart module should create a chart for the results of a metric run, the used metric configuration for that run and a chart configuration.

Design Description

The chart module is accessed through the `ChartFacade` (see Figure 6.17). The `ChartFacade` contains the method `createChart` that takes an XML element as input and returns a chart object. It uses the charting library JFreeChart [JFR] version 1.0.3 for the creation of the charts.

The XML element contains the metric run element returned as result from the core module and a chart configuration. The chart configuration element allows the following parameters to be configured:

- Title. The title text for the resulting chart can be configured freely.
- Different charts. The resulting chart can be composed of multiple child charts. For each subchart, the following properties can be configured:
  - Displayed calculation results. The calculations for which results should be shown in the subchart can be specified.
  - Range axis label. The text for the range axis label can be specified freely.

  - Chart type. The subchart can be a line chart or a stacked area chart.

- Domain axis markers. Different markers on the domain axis, for example to display release dates, with different labels and dates can be specified.

Design Reasons

There are several reasons for the chosen design.

JFreeChart is used because it is a flexible, free, powerful and stable library for charting. The effort for creating charts using JFreeChart compared to the development of a custom charting solution is significantly lower.

The facade hides the implementation details of the charting module from its clients. Therefore, it can be restructured easily without affecting its clients. The facade also provides an access point for stable module testing.

The usage of an XML element as parameter provides independence from the core module. The data for the charting module might come from other metric tools, too, as long as it conforms to the expected format. Furthermore, the usage of an XML element as parameter provides a stable interface in terms of the Java method interface. The method can provide backward compatibility for future extensions by adding default values for additional parts of the XML element, and the clients are not affected because the Java method interface remains stable.

6.16 Frontend

Problem Description

A preliminary frontend to the core and chart module should be provided that enables users to evaluate metrics and display the metric results in charts. The programming effort should be minimized, because the main focus is on the core module.

Design Description

The frontend is implemented as Servlet using the Java Servlet API. It is a subclass of `HttpServlet` that evaluates the contents of a HTML form pro-

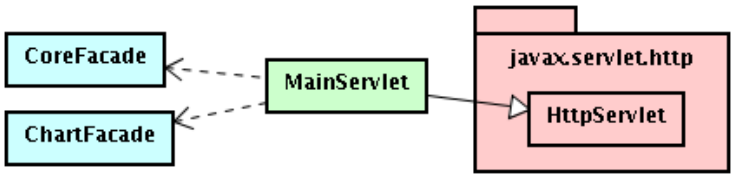

Figure 6.18: *Design of the web frontend*

vided by a simple HTML page. During the evaluation, it uses the `CoreFacade` and the `ChartFacade` to access the core and chart modules.

The form contents sent to the Servlet are mainly XML configuration data that is forwarded to the other modules.

Design Reasons

The implementation as a web frontend was chosen as solution for the following reasons:

- Programming effort. The programming effort for a web frontend solution is much lower than a GUI solution. A simple command line interface requires an even lower programming effort, but it has other disadvantages.
- Administration effort. The administration effort for a web frontend solution is lower than the administration effort for a GUI or command line solution. This is because only one installation must be maintained, the users access the web frontend with a web browser. In the other cases, there must be an installation on each client computer.
- Tool integration. Other tools can access the web frontend using HTTP and further process the results. In the GUI solution, such an interaction is hardly possible. In the command line solution, this is a distribution and maintenance problem.

The web frontend was implemented in Java because the main interfaces of the core and chart module are implemented in Java.

Chapter 7

Evaluation

In this chapter, the software process, the developed software product and some metrics that were calculated using the product are evaluated.

7.1 Software Process Evaluation

The evaluation of a software process by comparison to other processes is difficult, because the project, the participants and the circumstances are unique. Therefore a comparison to the same project executed with a different software process is often impossible. In the context of this project, this was the case.

Another way of evaluating software processes is using a standardized model like CMM/CMMI or SPICE [LL06, Chapter 11]. These models were developed to measure the software process maturity in companies and are therefore not applicable here.

Because the first two approaches to software process evaluation are not really applicable here, I have chosen a third way. The evaluation that follows is based on the process classification and process characteristics described by Sommerville [Som04, Chapter 28].

7.1.1 Process Classification

Sommerville describes four overlapping categories of processes [Som04, p. 670]: informal, managed, methodical and improving processes.

As the process used in the project has been chosen by the developer and

there is not strictly defined process model, it is an informal process and thus not managed. Furthermore, it is neither methodical, because no CASE tools for design and analysis have been used, nor improving, because there were no improvement objectives.

According to Sommerville, informal processes are an appropriate choice for small-sized systems like the product developed in this project.

7.1.2 Process Characteristics

Sommerville describes eight process characteristics [Som04, p. 667] which are evaluated in the following. A general problem of the evaluation that should be considered is the fact that small projects like this one depend more on people quality than on process quality. Therefore, the process evaluation might be influenced by the people quality, especially when it come to robustness and rapidity.

The process characteristics understandability, acceptability, reliability and maintainability are not applicable here because the software process was carried out by one person.

Visibility. The first two iterations produced requirements specifications, and the other iterations produced milestones releases. Therefore, a visibility on this high level is given. On a lower level, the visibility is rather limited. Each functional metric requirement from the specification is implemented by a test case, therefore a certain visibility is provided here, too. But there is no further visibility on this level. Therefore, the overall visibility is on a low to medium level.

Supportability. Although the complete development process from requirements engineering to deployment is not supported by CASE tools, such tools were used to a certain extent. In the implementation and rollout iterations, CASE tools like an IDE, refactoring tools, unit testing tools, build tools, a revision control system and UML tools were used. In fact, the development process would be too inefficient to be useful without the use of refactoring and unit testing tools. Therefore, the supportability is on a medium level.

Robustness. There were some unexpected problems during the development process. The requirements changed: the requirement for an advanced user interface was dropped and replaced by the requirement of an XML interface. Furthermore, serious system performance problems were discovered

while testing the second milestone on real data. The process provided the frame for dealing with these problems in a controlled manner. The problems were resolved quickly. Therefore, the robustness of the process is high.

Rapidity. The rapidity of the process was perceived by the developer to be fast. But because there is no comparison available of how fast other processes would have performed under the same circumstances, there is no objective data that supports this perception. Furthermore, it is estimated that about fifty percent of the development time was used for refactoring and restructuring. It is unknown to what extent this is relativized by the reduced amount of time spent on design, analysis and documentation and how this activity distribution affects the product quality. Therefore, a well-founded evaluation of rapidity is not possible.

7.2 Software Product Evaluation

In this section, the software is evaluated according to the criteria in the ISO 9126 norm [Bal98]. The criteria defined in the ISO norm were preferred to the model provided by Boehm [BBL76] because it is an official, standardized norm.

7.2.1 Functionality

Suitability. The metrics that were defined in the requirements specification are all covered by tests on specification level. The functional requirements for the chart module are covered by tests, too. That way, it is ensured that the functional requirements from the specification are implemented. The requirements engineering itself was a combination of domain analysis, an evaluation of existing tools and meetings with future users. It was revised several times until all stakeholders were satisfied with the set of requirements.

Accuracy. The metrics that were defined in the requirements specification are all covered by tests on specification level. Therefore, the agreed results and the precision of the calculations were checked. Furthermore, the software was tested against real data and the results were presented to and discussed with the future users. The results seemed plausible.

Interoperability. The software was designed to work on the data provided by Bugzilla based on the SQL interface of the Bugzilla database. In the other direction, it provides XML interfaces that can be called over HTTP. That

way, it is easy for other tools to use the functionality provided by the software. The other tools are not restricted to certain languages or frameworks.

Compliance. This criterion is not applicable here, because there are no official standards or regulations in laws that are relevant for the developed tool.

Security. Because the software is installed on a Kisters-internal server, only Kisters employees can use it. This was the only security constraint, each Kisters employee should be able to use the full functionality of the software. Therefore, all security requirements are fullfilled.

7.2.2 Reliability

Maturity. The software has not been in production use yet. Although the frequency of failures is not expected to be high, because the system was developed using a test driven implementation of the specification and has been tested with real metrics on real data, some failures will probably occur.

Fault tolerance and recoverability. The software handles each request separately and does not manage persistent data. This behavior is supported by the Servlet architecture that handles each request in a separate thread. If there are errors in the processing of one request, the other requests are not affected, and the Servlet remains responsive.

7.2.3 Usability

The tool will probably be used as a service or library in the future. Other tools will communicate with it through its interfaces, and direct user interactions are less likely. Therefore, the usability of the software was considered to be less important than the functionality, efficiency and maintainability. Thus, the following sections deal with the usability of the XML interface.

Understandability and Learnability. The basic logical concept of the application is a simple data processing concept. It takes a metric specification as input and calculates the result as output. The concepts of the XML interfaces for the metric specification and the results are well structured and do not overlap as far as possible. But because it has not been used by many other people, the understandability and learnability is quite unknown yet and still needs to be verified.

Metric	Total	Mean	Maximum
Total Lines of Code	5453		
Method Lines of Code (avg/max per method)	2158	3.18	31
Weighted Methods per Class (avg/max per type)	941	5.47	46
Number of Methods (avg/max per type)	643	3.73	30
Number of Classes (avg/max per package)	172	8.19	19
Number of Interfaces (avg/max per package)	24	1.14	5
Number of Packages	21		

Figure 7.1: *Size metrics for core module*

Operability. The software is executed as web application in a web application container that supports the Java Servlet Specification 2.3 [SER] like Apache Tomcat [TOM] on a server. Thus, the effort for operability is rather low.

7.2.4 Efficiency

Time behavior. The time behavior of the software was optimized (see Section 6.13) after using it on real data for the first time. On the test system, the results for metrics were returned within ten to twenty seconds on average, with a maximum return time of less than one minute. As it is expected that the metrics are not likely to be calculated in parallel, this is acceptable. Because the production system is faster than the test system and the test system contained a recent snapshot of the production system data, it is expected that the system is faster when put to production use.

Resource behavior. The resource behavior of the software was optimized (see Section 6.13) after using it on real data for the first time. In a default configuration of the Java Virtual Machine, no resource problems were discovered.

7.2.5 Maintainability

Analyzability, changeability and stability. These qualities are mainly based on the structure of the software. The use of many software design principles was already discussed in Chapter 6. In the following, a metric evaluation of the core module is presented. The chart and web frontend modules have less than 250 lines of code each, so such an evaluation does not make sense for them.

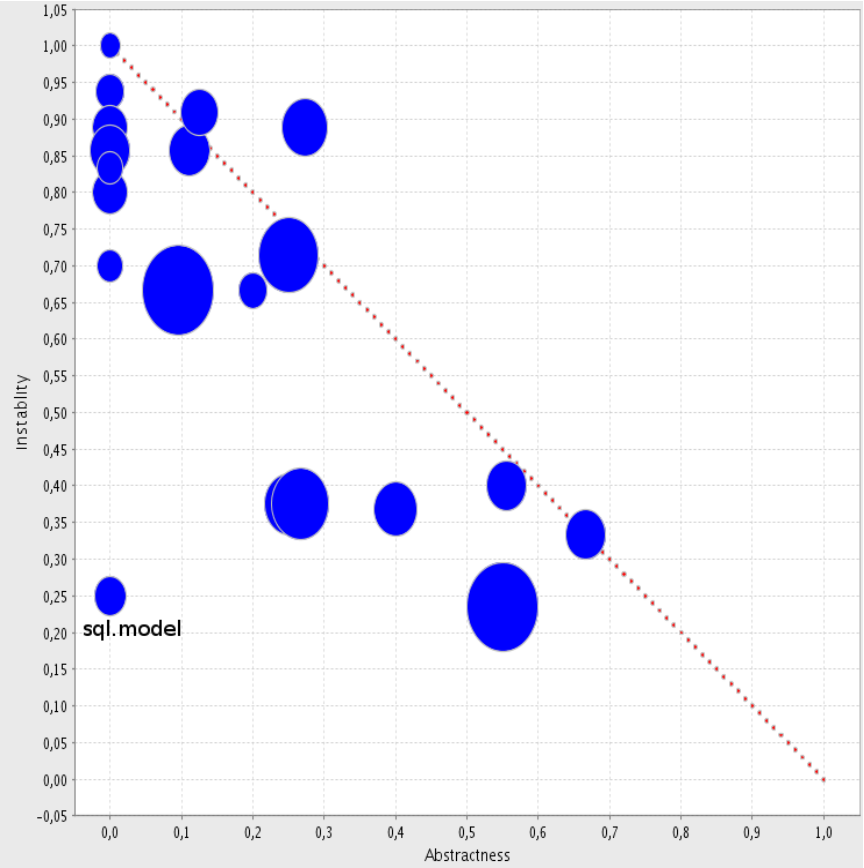

Figure 7.2: *Distance graph of core module packages*

The basic size metrics (see Figure 7.1) show that the code is modularized well. The methods, classes and packages are small in the average, and the maximum values are still acceptable. The weighted methods per class metric [HS96, p. 127], which is a size measurement that is more language independent than the lines of code, shows that the class and method complexity is rather low. The modularization into small, understandable units improves the analyzability, changeability and stability of the code.

On the package level, there are no cycles between packages. According to Robert C. Martin, the packages should be grouped near to a main sequence when analyzed on instability and abstractness, with unstable, concrete packages depending on the more stable, abstract packages [Mar94]. Figure 7.2 shows the different packages according to their instability and abstractness, although the package dependencies are not shown. Their distance from the main line is rather limited. The only package that has a big distance is the

`sql.model` package, and that is because it only contains constants describing the database layout. The dependencies are mostly from unstable to stable packages. An exception are the `util` and `sql` packages, which are referenced from more abstract and stable packages. Overall, the directions of the package dependencies and distribution of the packages in the distance graph indicates good analyzability, changeability and stability.

Testability. Many of the arguments from the previous paragraphs also apply to testability. Furthermore, because the system was developed in a test driven manner, there are some hooks for advanced testing, and the testability is height in general.

7.2.6 Portability

Adaptability and Installability. The software is deployed as a web application for the Java Servlet Specification 2.3 [SER]. It is therefore easily installable on every web application container that supports this specification.

Conformance. The software is implemented as a web application according to the Java Servlet Specification 2.3 [SER] and was written in the programming language Java for the J2SE 1.4.2 [JAV]. Therefore, it can be deployed on each platform that is supported by Java and into each web application container that supports the Java Servlet Specification 2.3.

Replaceability. The communication of other tools and the user with the software is based on HTTP and XML. The XML interfaces are specified by XML schema files and tests on specification level.

7.3 Metric and Usage Evaluation

Discussions of the metrics and charts created by the tool with the users gave a first impression of the advantages and problems of the tool and its usage.

One important advantage is that vague assumptions about certain developments are now supported by concrete figures. For example, the assumption of a constantly increasing workload was supported by the totals metrics and the remaining estimated workload metric.

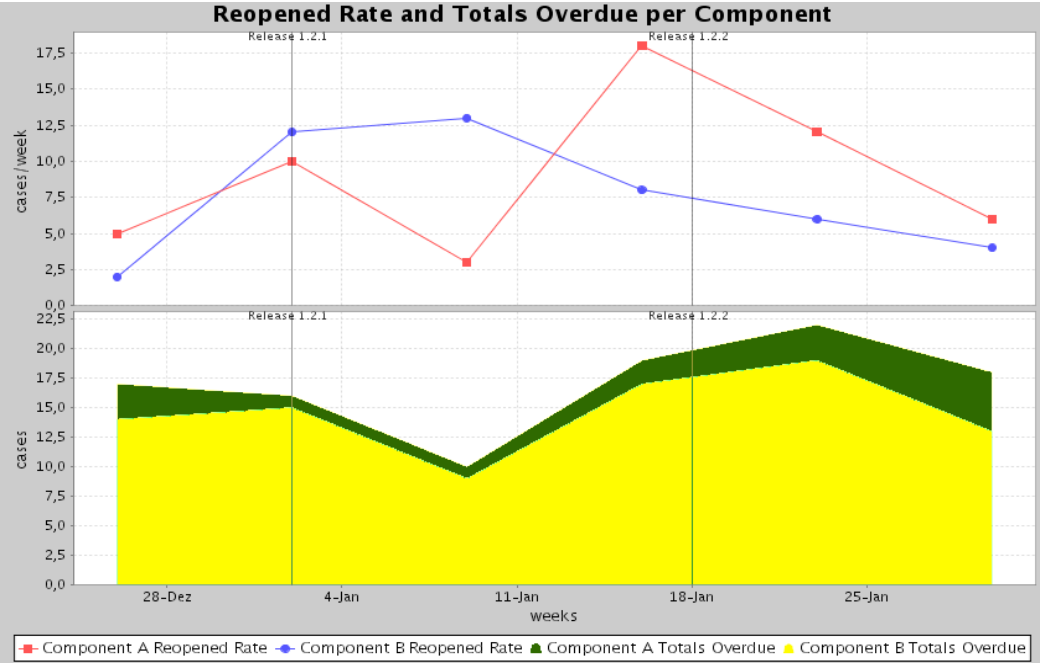

Figure 7.3: *Example output of the chart module*

Furthermore, the results from the tool can be used to control certain improvements, for example the falling rate of cases without original estimate and the falling rate of bugs without target milestone.

Another insight was that the metrics cannot be taken into account without further interpretation. Certain actions like a cleanup of the cases in the Bugzilla database have a profound impact on the metrics and without the knowledge of such events, misinterpretations are likely.

It should be noted that the figures themselves were not that useful without being displayed in charts. The visualization provided the foundation for an easy and efficient interpretation (see Figure 7.3).

Furthermore, there were some problems related to the metrics themselves that occurred during the first evaluations. They are described in the following:

Problems comparing incoming and outgoing rate

There are some problems comparing the incoming and outgoing rate, which then as consequence affect the backlog index.

If the incoming rate is defined as the number of cases created in a time interval, and the outgoing rate is defined as the number of cases resolved in a time interval, there are the following problems:

- Cases can be resolved multiple times if they are reopened
- Cases that are created outside the scope of the base filter and moved into its scope can be resolved and counted in the outgoing rate, but are not counted in the incoming rate. An example for this is the scenario when cases are filtered for a given product, but some cases were created in another product and then moved into the filtered product.
- Cases that are created inside the scope of the base filter and moved outside its scope are counted in the incoming rate, but not in the outgoing rate

This problem can be solved by changing the metric specification to take care of all these effects. For example, cases moved outside the scope of the base filter could be added to the outgoing rate.

Problems selecting the granularity of time periods

If the granularity of the time periods is too small, metrics that do not sum, but calculate the number of events in a time period can be too volatile to allow a good interpretation. Examples of this problem are the incoming and outgoing rate with a time period granularities like week or day, depending on the size and activity of the analyzed product or component.

If there are only a few events in a time period, their amount is mostly random. But if the time period is large enough, the values of the time periods become comparable. In my opinion, this is related to the law of large numbers [Hüb03, p. 116].

Problems with the maximum and average values

Often, some cases exhibit extremely large and uncommon values. This may affect metrics which calculate average and maximum numbers over the corresponding case attributes. For example, the age of a case can be really large

for some cases in a group, because they are regarded as unimportant and lay dormant for a long time. This leads to a high maximum, and more importantly, such extreme values affect the average. Therefore, it is recommended to prefer the median over the average value under such circumstances.

Problems with the backlog management index

The management backlog index (BMI) is defined as [Kan95, p. 105]:

$$\text{BMI} = \frac{\text{outgoingRate}}{\text{incomingRate}}$$

When the result is rendered in a chart with a linear scale on the range axis, the interval lengths for differences on the range axis are not comparable. For example, if the outgoing rate is twice as big as the incoming rate, the BMI is 2, whereas in the opposite case, the BMI is 0.5. If one compares the differences from the equal value 1, the difference is 1 in the first case and 0.5 in the second. This makes interpreting the charts more difficult.

One solution to the problem is the use of a logarithmic scale, and the another one is a different definition of the BMI:

$$\text{BMI}' = \frac{\text{outgoingRate} - \text{incomingRate}}{\text{outgoingRate} + \text{incomingRate}}$$

That way, the backlog index is normalized and returns values within the interval $[-1, 1]$. But is was perceived as unintuitive if the BMI is above zero if the backlog decreases and below zero if it increases. A third variant of the BMI solves this problem:

$$\text{BMI}'' = \frac{\text{incomingRate} - \text{outgoingRate}}{\text{incomingRate} + \text{outgoingRate}}$$

This variant is even closer to the definition of the backlog. Often, it is desireable to use the original differences instead of the normalized ones, which is the definition of the backlog itself:

$$\text{backlog} = \text{incomingRate} - \text{outgoingRate}$$

The advantage of using the original values instead the normalized ones is that one sees directly how the backlog affects the totals.

Chapter 8

Summary and Perspective

The goal of the project was the development of a software tool for the evaluation of change requests.

The requirements engineering consisted of four basic activities: an evaluation of the existing tools, a domain analysis, meetings with the stakeholders and the creation of the requirements specification.

At first, I evaluated four existing change request management tools. I analyzed them with respect to their capabilities in change request evaluation. Their similarities and differences were summarized and used as input for the domain analysis.

The other three activities were executed in an iterative way. The domain analysis consisted of the creation of concept maps and a glossary. The results of the domain analysis were used as input for the meetings, and the results of the meetings were used to revise the domain concepts and to create a requirements specification.

The result of the requirements engineering was a first scheme of the algorithm and a clear separation of concepts like case filtering, case value calculation, group value calculation and case grouping. Furthermore, many concrete metrics and the basic constraints like the choice of the programming language were specified.

The tool was then implemented in an iterative way. After the first two implementation iterations, the tool was tested and optimized on real data and some preliminary charts for some metrics were created. In the next iterations, the chart component and the web frontend were added and the robustness and documentation of the tool was improved.

The first impressions of the usage of the calculated charts and metrics confirmed the assumption that the tool can be useful for assessing the current state of products and projects with respect to the remaining workload, the process speed and quality and the product quality. Furthermore, it showed that the interpretation of the metric results needs to be done carefully, and that the specification of the metrics has to be done carefully, too.

In the evaluation part of the thesis, the tool was analyzed according to the ISO 9126 norm and the requirements specification. Additionally, the software process was evaluated, too.

There are several possibilities for future extensions to the tool. They are listed in the following:

- A user interface that is easy to use could be created. This is important for the acceptance and the use of the tool. The problem is displaying and editing the complicated metric specification in a user-friendly way.
- A link between Bugzilla and version control systems like CVS or Subversion could be established. The tool could then be extended to evaluate metrics that use data from the version control systems like the number of changed lines for certain cases.
- The calculation of metrics that start at different time points, for example to compare different releases, could be added.
- The tool could be wrapped as a web service to simplify the interaction with other tools.
- There could be a link from the chart to the cases that were used in certain groups. For example, clicking into the chart could open a new window containing the relevant cases and their links to Bugzilla. But it should be considered that for more complex metrics, this can be problematic because their values are not directly derived from the cases, but calculated, and therefore their link to certain case is less obvious and can be misleading.

- The chart module could be further extended. The formulas of the group value calculations could be displayed in the charts. The units on the range axis of the chart could be calculated automatically. Valid ranges could be marked in the charts by providing a mechanism for the specification of range markers. The charts could actively display the values of the lines when moving the mouse cursor over them.

To sum it up, a flexible tool for the evaluation of change requests was created. It offers the possibilities of putting it into production use and using the metrics calculated by it in the context of process and product evaluation and improvement in a real world scenario.

Appendix A

Used Software

- **Java Development Kit 1.4.2.13**
  Compiler and Platform
  `http://java.sun.com/j2se/1.4.2/`
- **Eclipse 3.3 Milestone 4**
  Integrated Development Environment
  `http://www.eclipse.org`
- **Subversion 1.4.2**
  Revision Control System
  `http://subversion.tigris.org/`
- **MySQL 4.1 Community Edition**
  Database
  `http://www.mysql.com/`
- **Apache Tomcat 5.0.28**
  Servlet container
  `http://tomcat.apache.org/`
- **Apache Ant 1.6.5**
  Build Tool
  `http://ant.apache.org/`
- **JFreeChart 1.0.3**
  Charting Library
  `http://www.jfree.org/jfreechart/`
- **Java Servlet API 2.3**
  Servlet Library
  `http://java.sun.com/products/servlet/`

- **Apache Log4J 1.2.13**
  Logging Library
  `http://logging.apache.org/log4j/`
- **Apache Xerces2 Java Parser 2.9.0**
  XML Processing Library
  `http://xerces.apache.org/xerces2-j/`
- **JDOM 1.0**
  XML Processing Library
  `http://www.jdom.org/`
- **JUnit 3.8.2**
  Test framework
  `http://www.junit.org/`
- **CAP 1.2.0**
  Source Code Analyzer Plugin for Eclipse
  `http://cap.xore.de/`
- **TagSEA 0.5.0**
  Source Code Navigation Plugin for Eclipse
  `http://tagsea.sourceforge.net/`
- **Sysdeo Eclipse Tomcat Launcher plugin 3.2 beta2**
  Apache Tomcat Launcher Plugin for Eclipse
  `http://www.sysdeo.com/eclipse/tomcatplugin/`
- **Subversive 1.1.0**
  Subversion Plugin for Eclipse
  `http://www.polarion.org/index.php?page=overview&project=subversive`
- **Checkstyle 3.5.0**
  Coding Conventions Checking Plugin for Eclipse
  `http://eclipse-cs.sourceforge.net/`

Bibliography

[APA] Apache Lucene. `http://lucene.apache.org/`. [Online, accessed January 26, 2007].

[ATL] Atlassian Software Systems Pty Ltd. `http://www.atlassian.com/`. [Online, accessed January 26, 2007].

[Bal98] Helmut Balzert. *Lehrbuch der Softwaretechnik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum, Akademischer Verlag, 1998.

[BBL76] Barry W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE*, pages 592–605, 1976.

[Bec00] Kent Beck. *Extreme Programming Explained - Embrace Change*. Addison Wesley, 2000.

[BUGa] Bugzilla. `http://www.bugzilla.org/`. [Online, accessed January 26, 2007].

[BUGb] The Bugzilla Guide. `http://www.bugzilla.org/docs/`. [Online, accessed February 11, 2007].

[Buh04] Axel Buhl. *Grundkurs Software-Projektmanagement*. Hanser, 2004.

[COD] Code Beamer. `http://www.intland.com/products/codebeamer.html`. [Online, accessed January 26, 2007].

[DeM95] Tom DeMarco. *Why Does Software Cost So Much? And Other Puzzles of the Information Age*. Dorset House Publishing Company, Incorporated, 1995.

[FP96] Norman E. Fenton and Shari L. Pfleeger. *Software Metrics. A Rigorous and Practical Approach*. PWS Publishing Company, Boston, MA, 1996.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1994.

[HS96] Brian Henderson-Sellers. *Object-oriented metrics. Measures of complexity.* Prentice-Hall, 1996.

[Hüb03] Gerhard Hübner. *Stochastik. Eine anwendungsorientierte Einführung für Informatiker, Ingenieure und Mathematiker. 4. Auflage.* vieweg, 2003.

[INT] Intland GmbH. `http://www.intland.com/`. [Online, accessed January 26, 2007].

[JAV] Java 2 Platform, Standard Edition (J2SE) 1.4.2. `http://java.sun.com/j2se/1.4.2/`. [Online, accessed February 16, 2007].

[JFR] JFreeChart. `http://www.jfree.org/jfreechart/`. [Online, accessed January 31, 2007].

[JIR] JIRA. `http://www.atlassian.com/software/jira/`. [Online, accessed January 26, 2007].

[Kan95] Stephen H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley, Reading, MA, 1995.

[LEV] Wikipedia: Levenshtein Distance. `http://en.wikipedia.org/wiki/Levenshtein_Distance/`. [Online, accessed February 17, 2007 14:26 CET].

[LL06] Horst Lichter and Jochen Ludewig. *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, 2006.

[LW00] Dean Leffingwell and Don Widrig. *Managing Software Requirements.* Addison-Wesley, 2000.

[Mar94] Robert C. Martin. OO Design Quality Metric. An analysis of dependencies. `http://www.objectmentor.com/resources/articles/oodmetrc.pdf`, 1994. [Online, accessed February 14, 2007].

[MOZ] The Mozilla Organization. `http://www.mozilla.org/`. [Online, accessed January 26, 2007].

[PGF96] Robert E. Park, Wolfhart B. Goethert, and William A. Florac. *Goal-Driven Measurement. A Guidebook.* Software Engineering Institute, Pittsburgh, PA, 1996.

[POLa] Polarion for Subversion. `http://www.polarion.com/subv/`. [Online, accessed January 26, 2007].

[POLb] Polarion Software GmbH. `http://www.polarion.com/`. [Online, accessed January 26, 2007].

[RU89] Dieter H. Rombach and Bradford T. Ulery. Improving software maintenance through measurement. *Proceedings of the IEEE*, 77(4):581 – 595, April 1989.

[Sed03] Robert Sedgewick. *Algorithms in Java. Parts 1-4: Fundamentals, Data Structures, Sorting, Searching. 3rd Edition.* Pearson Education, Inc., 2003.

[SER] Java Servlet Technology. `http://java.sun.com/products/servlet/`. [Online, accessed February 16, 2007].

[SHT05] Harry M. Sneed, Martin Hasitschka, and Maria-Therese Teichmann. *Software-Produktmanagement. Wartung und Weiterentwicklung bestehender Anwendungssysteme.* dpunkt, 2005.

[Som04] Ian Sommerville. *Software Engineering 7.* Addison Wesley, 2004.

[Tha97] Richard H. Thayer. *Software Engineering Project Management, 2nd Edition.* Wiley, 1997.

[TOM] Apache Tomcat. `http://tomcat.apache.org/`. [Online, accessed February 16, 2007].

[Tuf98] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, USA, 1998.