Tool Support for Continuous Quality Assessment

Florian Deissenboeck, Markus Pizka, Tilman Seifert*
Software & Systems Engineering
Technische Universität München
85748 Garching, Germany

Abstract

Maintenance costs make up the bulk of the total life cycle costs of a software system. Besides organizational issues such as knowledge management and turnover, the long-term maintenance costs are largely predetermined by various quality attributes of the software system itself, such as redundancy and adequate documentation. Unfortunately, many quality defects can hardly be corrected retrospectively after they have penetrated the system. A much more promising approach than correction is to avoid decay and to preserve a constant high level of quality through a continuous real-time quality controlling process. To reduce the costs for the required frequent quality assessments, adequate tool support is indispensable. This paper proposes to integrate measurement tools into a flexible and extensible yet high performance quality assessment tool. We present the design and implementation of this tool and report on our experiences made with it in a medium-sized academical project. Among the positive effects are improved software product quality and reduced efforts for manual quality assessments as well as increased awareness for quality issues.

1. Introduction

Software maintenance activities typically consume 80% of the total cost of a software system [3]. While 80% sounds dramatic, the interpretation of this number, its reasons and consequences are not as obvious as they are often conceived. For example, rather low annual maintenance costs of 10% of the original development costs over a period of 30 years sum up to 75% over the complete lifecycle. Hence, 80% by itself might not indicate a problem but be a side-effect of the long-term success of a software system. Likewise, spending 75% of the budget on maintenance activities [19] does not justify the term "maintenance crisis" without a detailed view on the actual situation and context. However, it is clearly a problem if maintenance activities, such as adding new functionality, consume excessive amounts of time, or if change requests are primarily corrective instead of perfective or adaptive ones [17].

Besides organizational issues, e. g. qualification and turnover, the major reason for such undesirable effects are quality defects. Standards such as ISO 9126 [12], define *maintainability* by means of a set of quality attributes, such as *analyzable, changeable, testable, and stable*, or more detailed ones such as consistent and concise naming [7].

Violations of these quality criteria may either be introduced during initial development or caused by long-term decay [9]. Once these defects have found their way into the system, they are usually very hard to correct. Obviously, restructuring a weak architecture requires extensive resources but even assumed minor changes, such as the removal of copied and pasted code duplicates[1] can easily become excessively complex.

Reel states that "by the time you figure out you have a quality problem, it is probably too late to fix it" [18]. Therefore, we claim that it is necessary to continuously and closely monitor the quality of software systems in order to prevent defects as far as possible from creeping into the system. We argue that given appropriate tool support and a certain amount of process discipline, the cost of maintaining high quality can be diminishing low and will deliver rapid pay-off, often even within the development phase, already. The tool concept proposed in this paper is based on the seamless integration of different measurement tools into a flexible, extensible, yet efficient quality assessment architecture. This setup allows to assess the quality of a software system in real-time and paves the ground to establish a continuous quality controlling process.

Outline After a discussion of related work on quality measurement tools in Section 2 we will state a set of requirements for suitable tool support for real-time quality

*Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project VSEK (Virtual Software Engineering Competence Center).

[1] clone removal

controlling with respect to maintainability in Section 3. The design and implementation of our tool *ConQAT* (Continuous Quality Assessment Tool) is detailed in Section 4. In Section 5 we report on our experiences with *ConQAT*. Finally, Section 6 summarizes our findings and gives a glimpse on future work.

2. Related Work

2.1. Software Quality

There is plenty of work on software quality in general [13, 15, 16] as well as on more specialized topics like *maintainability* [1, 6]. In fact there is far too much previous work on the topic to be covered here in its entirety. This is especially true as different authors approach this ample topic from very different angles and deal with it on very different levels of abstraction using very different methods. Nevertheless nearly all previous work shares one common result: The great variety and diversity of factors on software quality. This is highlighted particularly by a number of publications about *software quality models* that aim at decomposing complex quality attributes like *maintainability* into more tangible ones. Examples are Dromey's model [8], the original *Factors-Criteria-Metrics* model [4] and our two-dimensional quality model presented in [5].

2.2. Software Quality Tools

Commercial vendors as well as the open source community offer a plethora of diverse software analysis and quality assessment tools.

An approach taken frequently is the construction of a *facts database* or an intermediate *meta-model* representing an abstraction of the source code. Commonly used levels of abstraction are for example an *abstract syntax tree* (AST) or a *call-* or *dependency-graph* of the system. Typically, object-oriented [14] or relational models [2] are used to implement the selected level of abstraction. Albeit great differences in the detailed design of the various *facts* or *meta-model* based approaches to software analysis, all of these approaches preprocess the input before performing the actual analysis. During this preprocessing stage the system under investigation gets parsed and transformed into the format of the meta-model. All analyses and assessments are then carried out on the meta-model.

Although this well-structured tool design is consequent and elegant from a software engineering point of view, it has a major drawback: it rigidly defines a certain level of abstraction and thereby limits the range of possible analyses. Code duplication checks, for example, can't be performed on the dependency graph. As important quality aspects are of very diverse nature and rely on different information this problem is typically circumvented by offering multiple layers of abstraction for the different types of analyses. But this means that all information needed to construct the complex multi-layer meta-model needs to be acquired for the entire system, which in turn renders building the meta-model a very expensive task. In practice, preparing the meta-model of a large scale system often takes several hours which is unacceptable for real-time quality assessment. In fact, these tools are used by quality experts for rather infrequent in-depth investigations of certain quality criteria. They are not suited for the integration into a continuous quality controlling process.

Besides *facts* and *meta-model* based approaches, there are numerous metric tools. They come as stand-alone tools or plug-ins for development environments like Eclipse (e. g. *Metrics*[2]). This range of products is supplemented by a number of assessment (or audit) tools like PMD[3] which usually offer batch and interactive operation modes, too. Some of the available tools are designed as extensible platforms which may be augmented with custom analyses, allowing a centralized view of the results. However, they fail to provide means to compose more complex analyses from a set of simple analysis modules — even though, literature on software quality [4, 12] clearly points out that quality is a complex and diverse matter which can only be assessed by analyzing and aggregating a great number of influencing factors. Therefore the composition of different analyses that create a holistic view on a system's quality is a crucial feature of a quality analysis tool.

3. Quality Management

The success of quality management depends on the quality management process, the criteria used to assess quality, and the tool support provided.

3.1. The Process

An ideal quality management process allows to detect small deviations from the target quality, so that corrective action can be taken immediately, and it allows to do so early and continuously, i. e. "in real-time of development". It provides and uses a set of information about the current quality status of the system. This information must be complete and detailed on one hand to be of direct value to the developers, on the other hand it must provide an aggregated view on the same information to give a quick and accurate overview over the status of the system. It must be possible to tailor this information to the specific needs of the project. The information must be up-to-date and available at any time; the collection of data must not affect development tasks.

[2] http://metrics.sourceforge.net/
[3] http://pmd.sourceforge.net/

3.2. The Criteria

Due to the diversity of the factors influencing product quality the real challenge in software quality management is to find the right criteria that allow to draw an accurate picture of the quality of a system. Identifying these criteria proves to be very complex as every project constellation demands a unique set of quality criteria that match its specific properties (in addition to generally accepted criteria).

Quality models like [4] or [8] help identifying the relevant criteria by providing a structured framework. In [5] we present an approach that allows advanced project-specific tailoring by taking into account the activities carried out within a certain constellation.

By not limiting itself to aspects that can be measured automatically this approach further differs from other quality management endeavors. We believe that many essential quality issues, such as the usage of appropriate data structures and meaningful documentation, are semantic in nature and can inherently not be analyzed automatically. Moreover the relevance of automatically determined metrics, e. g. *coupling between objects (CBO)*, is questionable if evaluated in isolation.

We claim that only the combination of different measures, including systematic "manual" evaluations as well as automatic measurements, can provide a coherent set of criteria that serve as indicators for the quality of the system. An example is cross-checking our *manual* source code rating (see sec. 5) with *automatic* checks for JavaDoc comments, comment ratio, unit tests results, test coverage etc.

3.3. The Tools

Following this reasoning, we derive the following key requirements for the tool support of the quality management process.

Static Output: The tool should work in a non-interactive, automated way with a static output so that there is no additional cost (in time, effort, or motivation) to use it. It should integrate different result types. For *ConQAT*, we decided to produce an HTML page in the nightly build that gives a detailed, integrated report about all quality attributes considered in the following.

Different Views: Information should be available in a detailed form as well as in an aggregated, brief form to satisfy the needs of different stakeholders in the project.

Flexibility: The system needs to be flexible in a way that it is easy to combine different analyzers and to configure different analysis runs that exactly match the needs of the project. In different phases and for different projects, different questions might be asked. The tool should be easily configurable to give concise answers.

Extensibility: The same argument leads to the requirement that the tool should provide an infrastructure to make extensions with new analyzers as easy as possible.

Diversity: Quality attributes can be discussed on many different levels. The tool should make no restrictions about the level of detail, the level of granularity, nor the type of the attribute of the analysis. Examples of analysis levels include the source code, the build process, the documentation, or repository properties.

4. ConQAT

To the best of our knowledge none of the tools available completely satisfies the requirements pictured above. We therefore designed *ConQAT* from scratch. Our design considerations are led by the requirements above and by the experiences we made with existing analysis tools as well as our own prototypes.

A central decision is not to follow the approach taken by fact-databases or meta-modeling tools as we believe that a common abstraction level inherently increases implementation effort and hampers performance. Why load a complex call graph representation of the entire source code into a relational database, just to find out how often it contains the string literal "TODO"? The usual argument in favor of a common abstraction level is reduced redundant analysis steps. As *ConQAT* shows, this problem can be elegantly circumvented by using smart caching mechanisms while making sure that no superfluous analyses are carried out.

It quickly became evident that our tool must provide a flexible extension (or plug-in) mechanism to fulfill our requirements. These extensions can carry out various analyses whose results should be composable.

The main challenge here was the design of an architecture which is rigid enough to allow the efficient combination of different analyses while being flexible enough to integrate the plethora of different kinds of analyses. Detailed analysis of different extensible architectures pointed out that there is in fact a *spectrum of flexibility*. However, there's always a trade-off between the flexibility and the expressiveness of the extensions.

On one end of the spectrum you find architectures which are extremely rigid. They define a very stringent extension interface and thereby limit the extensions' expressiveness. Nevertheless, they allow a flexible composition of the extensions and permit a rich infrastructure in the architectural core of the system. On the other end of the spectrum you find architectures which define a very unspecific interface and integrate their extensions only loosely. This enables extensions to be much more powerful but limits composition possibilities and inhibits a rich common infrastructure.

To obtain a better understanding of this spectrum we developed two prototypes close to both ends of it. The one on the *rigid* end basically supported a mapping from compilation units to numerical metric values. Obviously this

mechanism allows very efficient composition of different analysis modules but limits the range of analysis types. It doesn't support metrics which yield anything but a numerical value (without cumbersome workarounds) and makes analyses with a granularity different from compilation units impossible. The prototype at the other end of the spectrum was more or less a web portal which allowed the extensions to contribute HTML pages with their analysis results. It should be clear that this approach allows almost unlimited possibilities for the extensions but makes a meaningful composition of different extensions nearly impossible.

4.1. Design Considerations

Our analyses and experiments showed that finding the "right spot" on this spectrum was impossible due to the multifaceted nature of quality factors. Whenever we came up with a seemingly suitable set of interfaces a new requirement for a specific quality analysis revealed another deficiency. Though this could be attributed to a lack of skills on our part we are convinced the problem is caused by the great number of diverse analysis types a system like this must support.

We therefore opted for a solution which avoids picking a fixed spot on the *flexibility spectrum* and thereby limiting the system's versatility. Central idea of the selected solution is to specify interfaces that are general enough to support literally every kind of analyses and let evolutionary mechanisms work out more precise interface definitions for components that allow meaningful composition.

These considerations finally led to the design depicted in figure 1. The central element of *ConQAT*'s architecture are *processors* that are interconnected in a pipes-and-filter oriented style. These processors have highly diverse tasks and work like functions that accept multiple inputs and produce a single output. The *Driver* component is responsible for configuring the processor network and passing information from one processor to another. Processors may access external data like the file system or databases either directly or using one of the provided *Libraries*.

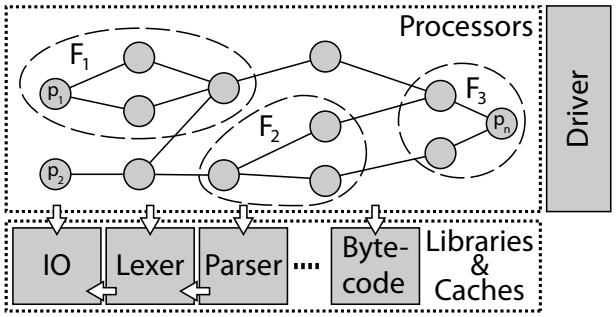

Figure 1. Architectural Overview

4.2. Analyses Composition

A simple example for composing an analysis of multiple processors is depicted in figure 2. Purpose of this analysis is to determine the average length of methods and to assess it with regard to a threshold. The analysis is composed of 7 processors which perform highly diverse and dedicated tasks. Processor *Scope* analyzes the file system and records the directory structure for all source code files that match a certain naming pattern. This tree-like data structure is forwarded to processors *LoC* and *#Methods* which determine the lines of code of each source file respectively compute the number of methods whereas the latter uses a parser or bytecode analyzer (provided as library). Processors *#Methods* and *LoC* both annotate the original data-structure with integer values describing the results of their analyses and hand them to processor *Div*. This is a very simple processor which solely computes the average method length for each source file. Processor *Assessment* assesses the average method length with regard to a predefined threshold and rates each source file on simple traffic light scale with either GREEN, YELLOW or RED. Processor *Aggregator* aggregates these assessments from the leaves to the root of the tree, i. e. nodes that have RED child nodes are themselves rated RED. Finally processor *Output* writes the results to a file with a suitable format like HTML.

Figure 2. Processor composition example

4.3. Agile Architecture Evolution

The type of data exchanged between processors is purposely unspecified to allow greatest possible flexibility. As *ConQAT* and its processors are implemented in Java it is actually defined as *java.lang.Object*. Nevertheless processors must define their concrete interfaces by means explained below. Our hope was that during the continuing extension of the tool *families of processors* with matching concrete interfaces would emerge like it is indicated in figure 1 by the dashed "clusters" denoted with F_i.

Processor Families Fortunately our assumption was confirmed very quickly: After implementing a couple of processors the desired *families* emerged. Examples are processors that perform calculations on scalar values as typically done when processing the results of metric analyses. These processors have no knowledge of the origin of the

values they process and can thereby be flexibly combined and reused in all situations that demand basic calculations.

Another example are processors that deal with the "traffic light assessments" we typically use. Besides assessing the results produced by other processors they are specialized in aggregating and filtering assessed results.

To actually analyze the system there is a family of processors that deals with the analyses of source code. This family can be subdivided in processors that analyze code in a language-independent way and processors that rely on more complex parsing mechanisms.

In addition to that there is a family of processors that is responsible for creating human readable output of the results. To make the results easily available to all project participants this is done in HTML.

Collaborative Interfaces Experience shows that interfaces within processor families remain stable after a certain tuning phase due to their relatively limited scope. This allows flexible organization of analyses by composing processors in different ways. An obvious example are the processors that deal with scalar values. By implementing a set of processors which perform basic calculations, more complex calculations can be performed by composing processors. Note that formally the computability expressed by composition is limited as we don't allow recursive calls to the processors. In practice this proves to be of no significance since each processors may implement every computable function.

Equally important are the interfaces between different families of processors. This is best illustrated by the following example. There is a family of processors that perform code audits like checking code format conventions or finding dubious pieces of code like empty blocks. These processors create lists with audit warnings for each source file. A simple interface between these processors and the ones described above is a processor that counts the number of warnings for each source file. This number may then act as input to further processors which perform calculations on it or assess it with regard to predefined rules. Here too, experience shows that the interfaces become stable relatively quickly.

Controlled Evolution By exploiting evolutionary mechanisms this approach leads to the modularization which we weren't able to design from scratch due to the great diversity of requirements. Evolutionary approaches demand measures of control to ensure success and avoid undesired developments. Problems that typically arise and which we experienced as well are "bloated" functionality of single processors and redundancy as two or processors implement the same functionality. We counter these effects with precisely the same continuous quality controlling measures we advocate in this paper. This involves clone detection, static checks for architecture violations combined with manual reviews. From the very beginning, we used *ConQAT* in a bootstrapping manner on itself to integrate these activities.

Central to these activities was identifying commonly used functionality and moving it to libraries that can be accessed by all processors.

Categorization The processors that evolved during *ConQAT*'s 10 month lifetime can be broadly categorized as follows:

Scoping. A processor may define the scope of particular analyses. It does this by building an appropriate object tree (e. g. representing files and directories) and passing it on to other processors.

Filtering. According to some filtering criterion, a filtering processor may remove particular nodes from the tree (e. g. discard files edited by a certain author).

Analysis. An analyzing processor carries out a particular analysis on the elements of an object tree and annotates the tree elements with the results (e. g. lines of code or number of comment lines); or it may analyze the whole tree (e. g. analyzing dependencies between its elements) and produce a new result type (like a dependency graph).

Aggregation. An aggregating processor collects values from different analyzers and annotates them with aggregated values (e. g. comment ratio).

Output. Finally a processor responsible for the output collects the analysis results and displays them in a human-readable format (e. g. HTML).

4.4. Configuration and Type Safety

This agile evolution of the architecture must be supported by a solid technical basis that inhibits uncontrolled growth of the interfaces. Typically one would expect that our decision to loosely specify the interfaces between the processors would result in a mess of explicit cast operation and the accompanying inevitable cast errors. Indeed the problems arising from the unspecified interfaces initially made our approach look infeasible. After implementing about 15 different processors we realized that the problems of non-explicit interfaces and the required explicit type casts introduced too many sources of errors to achieve a well maintainable system. We therefore developed a novel solution which we regard powerful and elegant. As we consider it essential for the success of *ConQAT*, this solution is presented in detail along with *ConQAT*'s configuration mechanism.

Configuration An important design decision affects the mechanism that allows users and extenders of *ConQAT* to configure composed analyses from simple building blocks

(the processors). In early prototypes this configuration was simply done by hard-coding the configuration with Java. As even the most simple reconfiguration of the system demanded modification of source code, re-compilation and redistribution of the whole system it became evident that this approach is not an option for a system whose central requirement is flexibility.

We therefore moved to a solution that employs a declarative (XML) configuration file to describe the interconnection of processors. This resembles the mechanisms typically used by extensible architectures like the Eclipse platform [11]. The disadvantage of this approach is the need for a minimal interface which processor implementations have to adhere to, and a mechanism to describe the mapping between the declarative configuration file and the implementation of the processors.

Processors' Interfaces As our processors are basically functions, their interfaces could be described by a single method:

```
Object process(Object[] parameters);
```

This interface precisely displays the problem discussed before: it is in fact untyped. As implementations can hardly perform any real work on objects with type *Object* they need further knowledge of the actual type of the parameter objects. Processor composition is further hampered by the fact that result type of a processor is unspecified.

With the new features of Java 5 the latter problem can be solved relatively easily using covariant return types. Covariant return types allow implementers of an interface to refine the return types of methods by using a subclass of the original return type, e. g.:

```
Integer process(Object[] parameters);
```

Unfortunately the former problem can't be solved as easily since covariant method parameters are unsafe and therefore not supported in Java. Central idea of our solution to this problem is to omit input parameters in the interface and leave their definition up to the processors. This is achieved by using Java's annotation mechanism[4].

Two example processor implementations are shown on the right hand side of figure 3. Both processors implement the parameterless method *process* and define their result types by using covariance. Processor *FileSystemScope* is responsible for scanning a given directory path for all Java source files and creating an *IFileSystemElement*-object that describes the resulting directory tree. The task of processor *LOCAnalyzer* is to annotate each leaf element of this tree object with the number of lines of code the corresponding source file has. Obviously this processor needs an object of type *IFileSystemElement* to work on. Therefore it defines a method

```
void setRoot(IFileSystemElement root);
```

and annotates it as a *@AConfigElement*. This annotation informs the *ConQAT* runtime system that the annotated method is meant to provide an input parameter. Additionally one may use the annotation to specify further details e. g. if this parameter is mandatory or not.

Type Safety To fully grasp the benefit of this approach one must understand how the connection of different processors is configured with *ConQAT*. A typical configuration file is shown on the left hand side of figure 3. It defines the processors named "source" and "loc-analysis" where the latter one is connected to the former one by referencing its name ("@source"). This connection is indicated by line ①. This configuration implies a corresponding connection between the implementation of the two processors as shown by line ②. Now the advantage of this mechanism becomes evident: *ConQAT*'s run-time system can make sure that two connected processors have matching interfaces by using Java's reflection mechanism. In fact *ConQAT* refuses to run an analysis if the interfaces of connected processors do not match. The advantage of this approach is that type safety needs be ensured only once before running the first analysis. We call this approach "configuration time type checking" (opposed to compile time or runtime type checking).

Besides this, the figure also shows that there is a defined mapping between the configuration file and the processors' implementations. Line ③ shows that the processor implementation is referenced by specifying the class name in the configuration file. Lines ④ and ⑤ exemplify how XML-elements are mapped to the corresponding input parameter methods: By using annotations, class *FileSystemScope* states that it expects only one configuration element called "input" that has exactly one attribute "root" with type *String*. In contrast to parameter "@source" which describes a reference to the output of the processor *source*, "src" is an immediate parameter since strings can be provided in the configuration file itself.

In fact annotations in processors are not only used to ensure type-safety and allow a defined mapping to the declarative configuration file but also provide a basis for automated generation of processor documentation.

Runtime The driver component is responsible for interconnecting the processors as defined in the configuration file and running the analysis. During start-up the driver loads the processors' classes via reflection and uses their

[4] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

meta data to ensure type safety. It thereby ensures that the configuration is valid before starting the analysis.

Processors are topologically sorted and then ran one after another. During execution the driver passes the result from one processor to the next. If a processor's result is used more than once the driver is responsible for cloning it. It additionally performs some monitoring tasks to provide debugging information if one of the processors should fail.

4.5. Caching Mechanisms

By nature, several different processors work on the same resources, e. g. different code style analyses on the AST of the compilation units. *ConQAT* offers a set of *libraries* that provide commonly used functionality. Apart from the already mentioned parser library this includes libraries as simple as the IO library and as complex as a library that provides access to the system's call graph.

These libraries form a central point of entry to the analyzed system's artifacts and thereby allow the implementation of efficient caching strategies. As it is very likely that different processors will use e. g. the AST of a particular compilation unit, the AST will be cached for future use and needs to be built only once. All *ConQAT* libraries use caching mechanisms which greatly reduces analysis time. To further improve performance the libraries are built on top of each other (if reasonable). The parser library uses pre-cached tokens from the lexer library. All caches are implemented in a memory-sensitive way and support dynamic uncaching if the system is short of memory.

5. Experiences

Our group actively develops a number of different software engineering tools. *ConQAT* was used to maintain a steady quality level during the ongoing work on these tools. Here we report on the experiences we made with the maintenance of roughly 100 KLOC[5] Java code in a period of 10 month.

5.1. Project Environment

Though our project environment doesn't fully reflect an industrial one we consider it well-suited for the evaluation of a quality assessment tool. Analogous to many industrial projects we have to deal with varying levels of experience and programming skills on the side of the developers which are students and researchers. The team size of 14 (10 students, 4 researchers) corresponds to the team size typically found in industrial projects. Like many current industrial projects our project involves high risks regarding new technologies due to the rapid development of the Java infrastructure.

We additionally face a number of problems which are less common in industrial projects but provide a touchstone for software quality management. As typical students projects last about 3 to 4 months, we have short project phases and turnover is very high. Students almost always work on the project part-time and developers often work in a distributed fashion (in the lab, in the office, at home).

This project situation demands efficient means of quality management to ensure a constantly high steady quality level while optimizing the times researchers spend on the project. To achieve this, researchers require an instrument that allows them not only to monitor project progress but also enables them to continuously assess the quality of new developments as well as changes to the existing resources.

5.2. Controlling Quality with ConQAT

To provide such an instrument, we set up a fully automated build environment that builds, tests and analyzes the entire source code every night. Therefore the source code is retrieved from the SCM[6] system and analyzed with a *ConQAT* configuration that itself is stored in the repository, too. This allows developers to run the *ConQAT* analysis on demand on their local workstations. Results of the nightly analysis are published on a internal[7] website to be accessed easily by all developers.

Iterative Improvement As foreseen, quality criteria evolved during project progress. This is due to new insights in quality issues gained by experience but also because the object of investigation evolves. For example, we intensified the attentiveness for the build system as this decayed over time, and we introduced a number of quality criteria regarding the new language constructs introduced with Java version 5.

This was done by extending the *ConQAT* configuration in use and, if required, adding new processors dedicated to this issues. Up to now this resulted in a library of 57 processors.

Examples The following excerpt of existing processors shows that these processors support a great variety of different analyses besides commonly known metrics like *lines of code* (LOC), *comment ratio* (CR), *coupling between objects* (CBO), etc.:

- *ANTRunner.* This processor executes a set of targets in ANT build files and monitors build success.

[5] thousand lines of code

[6] Software Configuration Management

[7] see `http://www4.in.tum.de/~ccsm/conqat-demo/` for an example

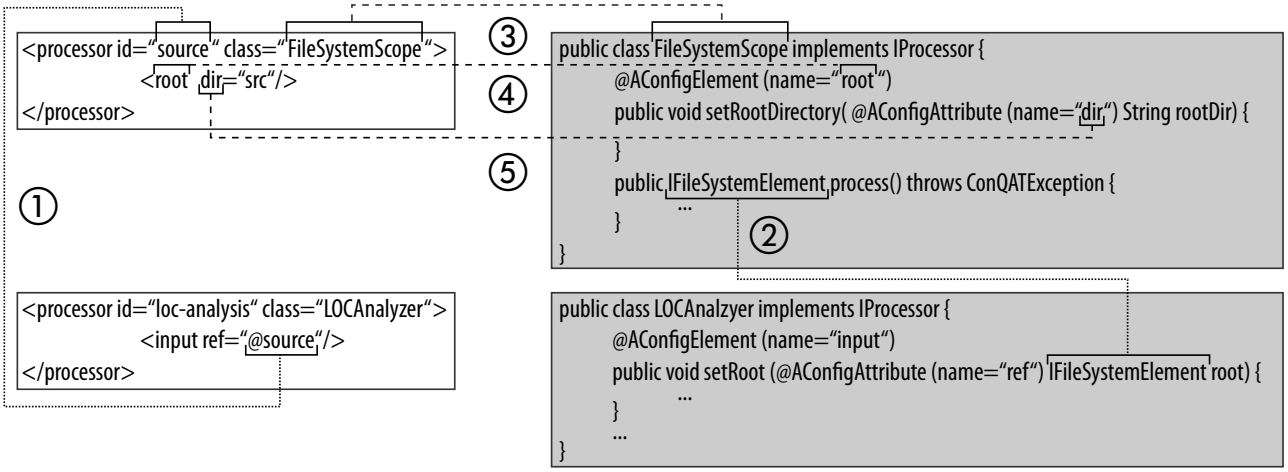

Figure 3. Mapping from Configuration File to Implementation

- *SVNLogMessageAuditor.* This processor checks if the commit log messages entered into the SCM system comply with our project guidelines.
- *JavaDocAuditor.* This processors checks if JavaDoc comments are present and comply to the guidelines.
- *JUnitRunner.* This processor runs unit tests and captures test outcome.
- *PMDRunner.* This processor acts as an adapter to the open-source tool kit PMD which offers a great number of source code audits. These audits are typically used to detect anomalies like empty code blocks and to control compliance to our coding conventions.
- *PerformanceMonitor.* This processor allows to run programs and capture performance characteristics. We use it to inhibit slow but hard to detect performance decline in performance critical parts of our programs.

Putting Results Into Relation An integral part of quality controlling management is the design of meaningful analyses. It is, e. g., of questionable use to assess the quality of a Java class on the ground of its mean method length. Nevertheless our experience showed that careful combination of different source values allows reliable and meaningful results in most cases.

For example a class that has 15 methods with an average length of 250 lines that do hardly comply to coding conventions and on top of it doesn't provide any inline or JavaDoc documentation is a valid candidate for a quality problem. Similarly a class that complies to every single project convention but contains the string "hack" might be another source of a quality problem.

ConQAT's architecture enabled us to express precisely this multi-faceted quality criteria by using its composition mechanism that allows us to put different results into relation and assess them accordingly.

Assessment It proved to be useful to perform these assessments on a simple ordinal scale with the three values RED, YELLOW and GREEN. Although this may seem too coarse-grained for some applications, our experiences show that it is satisfying in most cases and most times better than a complex scale which is difficult to read and therefore mostly ignored. Apart from that, the ordinal scale and appropriate aggregation mechanisms prevent typical mistakes when calculating metric values [10]. Nevertheless *ConQAT* offers more complex options, too. More sophisticated assessment mechanism can easily be realized when needed.

Figure 4 shows an excerpt of a typical assessment result. While files that have warning messages concerning their assessment rules are rated RED, tidy files are rated GREEN.

IUnit	G	
ISourceFile	R	Unused Private Field in line 41
SourceFile	G	
ChainedClone	R	Method putProperty(String, Object) has no JavaDoc comment. Method getProperty(String) has no JavaDoc comment. Method getPropertyNames() has no JavaDoc comment.
IPositionListAssociate	G	
CloneList	G	
Clone	R	Method putProperty(String, Object) has no JavaDoc comment. Method getProperty(String) has no JavaDoc comment. Method getPropertyNames() has no JavaDoc comment.
CloneClassComparator	G	
IUnitIterator	R	Immutable Field in line 59

Figure 4. Assessment Results

Review Cycles As software quality can't be completely determined by static code analysis, manual reviews are an integral part of our quality management activities. As reviews are inherently time-consuming we use *ConQAT* to

reduce review times as far as possible by using a combination of automatic and manual approaches.

For the manual evaluation, we use a simple three-valued scale to rate the review state of source code files: *premature*, *ready for review* and *accepted*. The rating is stored in the source code file itself by the author or the reviewer (depending on the state). By using information from the SCM we ensure that files that have been rated as *accepted* but underwent change subsequently get re-rated accordingly.

Obviously, we implemented a processor which extracts rating information from source files and maps them to the traffic light scale. Although it already proved to be handy to obtain a nicely formatted HTML output of the review state, we could really leverage *ConQAT*'s power by composing different analyses: With using existing processors, we easily created assessments that check if all files rated as *accepted* have full JavaDoc documentation and comply to our coding guidelines. This also includes identification of typical errors like missing default cases in *switch*-statements and known anti-patterns like god classes. Files not complying to these rules are displayed with rating RED.

As all files presented for review have a guaranteed minimal quality level this greatly helped to reduce the time spent on manual review activities.

Aggregation Our experiences showed that assessment data needs to be highly condensed to be of real use. Only if the key quality information that reflects an overall quality level can be viewed within a matter of minutes, quality controlling activities will actually be put into practice.

We therefore use a simple graphical aggregation for the quality assessments carried out by *ConQAT*. Figure 5 shows the main page of the assessment result website generated by the tool. For the sake of clarity the figure shows the results of some very basic analyses including a JavaDoc assessment and the manual rating described above. The colored bars to the left of the assessment description provide a quick overview of the results. So one can easily see that in project "CloneDetective" most files have rating status RED (for *premature*), while the majority of *ConQAT*'s source files has status GREEN (for *accepted*).[8]

This way, all important data is visualized on one page and easily accessible by the researchers (in the role of QA) as well as all other developers. More detailed information is literally only one mouse click away as the main page is linked to pages with detailed assessment results. Depending on the nature of the indicated quality problems, QA can present the evidence to the respective author or needs to check the source code for further examination.

[8]This is due to the fact that we introduced the code rating only after the development of "CloneDetective".

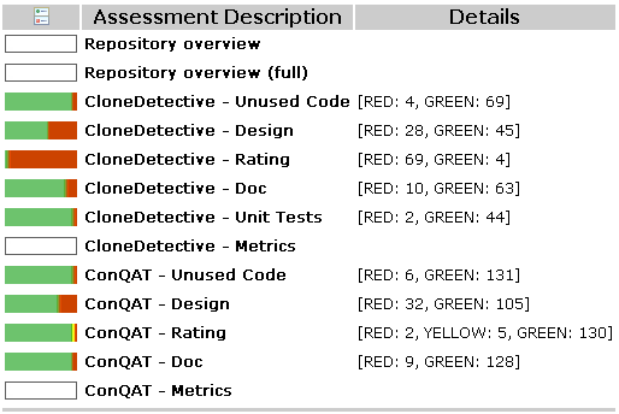

Figure 5. Condensed Assessment Data

5.3. Discussion

ConQAT's architecture proves to satisfy our requirements for flexible yet efficient quality analysis tools: It runs in a non-interactive manner and generates static HTML-output. It is flexible and extensible by offering two different levels of configuration; analyses can be composed using a declarative configuration file, and new analyses can be added by implementing new processors. The system's design does not limit analyses to a particular scope, granularity, or type of artifacts.

Experience shows that quality goals and criteria evolve over time. Obviously this requires a quality assessment tool that seamlessly supports this evolution. Therefore *ConQAT*'s flexible architecture proved to be crucial for a successful long-term quality controlling activities. This architecture allowed us to integrate even rather complex analyses like checks for unused code within a few hours. Of course this is facilitated by the Java community which offers an incredible rich variety of (open-source) analysis tools like parsers, coding convention checkers, test frameworks, etc.

In addition to that *ConQAT* offers convincing performance characteristics due to the heavy use of caching mechanisms. Even with a complex set of analyses including almost all of our processors our 100 KLOC repository could always be analyzed with matter of minutes. Experiments show that even analyses of systems as big as Eclipse ($\approx$ 2.5 MLOC[9]) can be carried out in similar times if the set of analyses is more restricted. These performance characteristics in combination with *ConQAT*'s flexibility provide a major advantage over meta-model-based systems if developers want to perform assessments of certain quality criteria during their daily work. They can easily limit the set of analyses by defining a trimmed down configuration file for *ConQAT* and get almost instant feedback for the desired analysis.

[9]million lines of code

The downside of the tool's design is a fairly complex configuration mechanism. Setting up a complete quality assessment configuration results in a configuration file of about 300 lines and requires thorough understanding of the processors involved. However, the aim was to develop a tool which is configured rather infrequently and run repeatedly with the same configuration (or a subset thereof). Besides that we expect the typical user to be an expert in the field and therefore don't consider this issue a problematic one.

6. Conclusions

While it is commonly accepted that software product quality is one of the key factors for project success, there is little common understanding of the factors influencing quality and their manifestation as product properties. Due to this blurry situation there is insufficient tool support for product quality assessments.

We claim that these fundamental shortcomings can be overcome to some extent by installing a quality management process that ensures that a set of project-specific quality criteria is controlled on a continuous basis. As such quality controlling activity is inherently costly, appropriate tool support is of paramount importance.

We deduced requirements for a tool to support this process and presented the quality assessment tool *ConQAT* which was designed in accordance to these requirements. Central property of *ConQAT* is its flexibility that allows to adequately support the immensely diverse tasks required for quality assessments. We presented the powerful extensible architecture that facilitates *ConQAT*'s flexibility and gave various examples for the application of *ConQAT*.

Up to now, we applied *ConQAT* in the development and maintenance of a tool collection at our department. As the development team of this tool collection (students and researchers) is subject to rapid turnover and the tool collection amounts to 100.000 LOC, sound quality management is *the* crucial factor for productivity.

What we observed so far is that the application of the tool greatly improved the consciousness for quality aspects as part of daily development tasks. Students feel comfortable with getting an automated feedback on their work before they actually submit it and state that this system indeed motivated them to create quality code from the beginning. Noticeable, it was also the first time that we heard students intensively discussing code quality issues in the lab. Besides that the main advantage is a reduction of time spent on quality measures on the side of the instructors while achieving and maintaining a product quality level not experienced before. This is especially interesting for the maintenance of our older tools.

While we can't quantify this statement, we are sure that this quality controlling process pays off rapidly, even for small projects (e.g. 6 students, part-time, 4 weeks, 20.000 LOC).

Naturally, *ConQAT* is far from being complete. Currently the top-most item on our agenda is adding database support to store analysis and assessment results. This would allow us to track evolution of the system's quality more conveniently. Besides that we plan to carry out a controlled experiment to quantitatively measure the effects of continuous quality management during the next lab courses.

References

[1] G. M. Berns. Assessing software maintainability. *ACM Communications*, 27(1), 1984.

[2] W. R. Bischofberger, J. Kühl, and S. Löffler. Sotograph - a pragmatic approach to source code architecture conformance checking. In *EWSA 2004*, pages 1–9. Springer, 2004.

[3] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[4] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.

[5] M. Broy, F. Deissenboeck, and M. Pizka. Demystifying maintainability. In *WoSQ 2006*. ACM Press, 2006. to appear.

[6] D. Coleman, D. Ash, B. Lowther, and P. W. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8), 1994.

[7] F. Deissenboeck and M. Pizka. Concise and consistent naming. In *IWPC 2005*, pages 97–106, Washington, DC, USA, 2005. IEEE Computer Society.

[8] R. G. Dromey. A model for software product quality. *IEEE Trans. Softw. Eng.*, 21(2), 1995.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan. 2001.

[10] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Softw. Eng.*, 20(3), 1994.

[11] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[12] International Standard Organization. *ISO 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use*, Dec. 1991.

[13] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1), 1996.

[14] A. Ludwig. *RECODER Technical Manual*. http://recoder.sourceforge.net, 2001.

[15] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In *WCRE 2004*, pages 192–201. IEEE Computer Society, 2004.

[16] J. McCall and G. Walters. *Factors in Software Quality*. The National Technical Information Service (NTIS), Springfield, VA, USA, 1977.

[17] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[18] J. S. Reel. Critical success factors in software projects. *IEEE Software*, 16(3):18–23, 1999.

[19] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. DoD, Mar. 1997.